Ensuring Lexicographic-Positive Data Dependence Graphs in the SIRA Framework

Sébastien Briais, Sid Touati, Karine Deschinkel

UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES

Ensuring Lexicographic-Positive Data Dependence Graphs in the SIRA Framework

Sébastien BRIAIS — Sid-Ahmed-Ali TOUATI — Karine DESCHINKEL

N° HAL-INRIA-00452695

February 2010

Rapport de recherche

Ensuring Lexicographic-Positive Data Dependence Graphs in the SIRA Framework

Sébastien Briais* , Sid-Ahmed-Ali Touati† , Karine Deschinkel‡

Thème : Optimisation de programmes
Équipe-Projet ARPA (laboratoire PRiSM) et ALCHEMY (laboratoire INRIA-Saclay)

Rapport de recherche n° HAL-INRIA-00452695 — February 2010 — 78 pages

Abstract: Usual cyclic scheduling problems, such as software pipelining, deal with precedence constraints having non-negative latencies. This seems a natural way for modelling scheduling problems, since instructions delays are generally non-negative quantities. However, in some cases, we need to consider edges latencies that do not only model instructions latencies, but model other precedence constraints. For instance in register optimisation problems, a generic machine model can allow considering access delays into/from registers (VLIW, EPIC, DSP). In this case, edge latencies may be non-positive leading to a difficult scheduling problem in presence of resources constraints.

This research report studies the problem of cyclic instruction scheduling with register requirement minimisation (without resources constraints). We show that pre-conditioning a data dependence graph (DDG) to satisfy register constraints before software pipelining under resources constraints may create cycles with non-positive distances, resulted from the acceptance of non-positive edges latencies. Such DDG is called *non lexicographic positive* because it does not define a topological sort between the instructions instances: in other words, its full unrolling does not define an acyclic graph.

As a compiler construction strategy, we cannot allow the creation of cycles with non-positive distances during the compilation flow, because non lexicographic positive DDG does not guarantee the existence of a valid instruction schedule under resource constraints. This research report examines two strategies to avoid the creation of these problematic DDG cycles. A first strategy is reactive, it tolerates the creation of non-positive cycles in a first step, and if detected in a further check step, makes a backtrack to eliminate them. A second strategy is proactive, it prevents the creation of non-positive cycles in the DDG during the register minimisation process.

Our extensive experiments on FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006 benchmarks show that the reactive strategy is faster and works well in practice, but may require more registers than the proactive strategy. Consequently, the reactive strategy is a suitable working solution for compilation if the number of available architectural registers is already fixed and register minimisation is not necessary (just consume less registers than the available capacity). However, the proactive strategy, while more time consuming, is a better alternative for register requirement minimisation: this may be the case when dealing with reconfigurable architectures, *i.e.* when the number of available architectural registers is defined posterior to the compilation of the application.

Key-words: Compilation, Code optimisation, Register pressure, Cyclic instruction scheduling, Instruction level parallelism

* Sebastien.Briais@prism.uvsq.fr
† Sid.Touati@uvsq.fr
‡ karine.deschinkel@iut-bm.univ-fcomte.fr

UFR des sciences
45 avenue des Etats Unis,
78035 Versailles cedex
Téléphone : +33 1 39 25 41 12 — Télécopie : +33 1 39 25 46 04

Garantir des graphes de dépendances de données lexicographique positifs dans la plateforme SIRA

Résumé : Les problèmes classiques d'ordonnancements cycliques d'instructions (pipeline logiciel) considèrent des graphes de dépendances de données avec des arcs à latences positives. Ces arcs représentent un modèle naturel des latences des instructions d'une boucle, qui sont généralement des quantités positives. Cependant, les arcs ne sont pas uniquement destinés pour représenter des contraintes de dépendances de flot de données, mais également pour représenter des contraintes de précédence diverses. Par exemple, une minimisation du besoin en registres dans un graphe de dépendances de données peut nécessiter l'insertion d'arcs à latences négatives si le processeur cible exhibe des délais d'accès aux registres en lecture ou en écriture (VLIW, EPIC, DSP). La présence des arcs à latences négatives ou nulles complique le problème d'ordonnancement d'instructions sous contraintes de ressources.

Ce rapport de recherche étudie le problème d'ordonnancement cyclique d'instructions avec minimisation du besoin en registres (sans contraintes de ressources). Nous montrons que la restriction d'un graphe de dépendances de données (GDD) pour satisfaire les contraintes de registres avant le pipeline logiciel peut créer des circuits à distance négative ou nulle. Un tel GDD est appelé graphe *non lexicographique positif* car il ne définit pas un tri topologique entre toutes les instances d'instructions; en d'autres termes, le GDD entièrement déroulé n'est pas un graphe acyclique.

Lors d'une approche constructive de compilation, nous ne souhaitons pas autoriser la création de GDD non lexicographique positifs, car il n'y a pas de garantie théorique pour l'existance d'un ordonnancement cyclique sous contraintes de ressources. Notre rapport étudie deux stratégies pour éviter la création des circuits à distances négatives ou nulles. La première stratégie est réactive, elle tolère en premier lieu la création des circuits à distance négative ou nulle. Une étape ultérieure de vérification teste l'existence d'un tel circuit, et corrige le problème en passant par un modèle dégradé (perte d'optimalité du modèle d'ordonnancement). La deuxième stratégie est proactive, elle élimine d'emblée les circuits à distance négative ou nulle sans perdre l'optimalité du modèle d'ordonnancement.

Nous avons effectué une batterie de tests intensifs sur les benchmarks FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006. Les résultats expérimentaux montrent que l'approche réactive fonctionne bien en pratique mais peut consommer plus de registres que l'approche proactive. Malgré une légère sur-consommation de registres, nous préconisons l'approche réactive lorsque le nombre de registres architecturaux est déjà fixé, ne nécessitant pas une minimisation du besoin en registres (où une consommation inférieure à la capacité du processeur suffit). En revanche, nous préconisons l'approche proactive dans un contexte de minimisation du besoin en registres; c'est le cas par exemple des architectures reconfigurables où le nombre de registres architecturaux est à fixer en postériori à la compilation de l'application.

Mots-clés : Compilation, optimisation de code, pression en registres, ordonnancement cyclique d'instructions, parallélisme d'instructions

Contents

Chapter 1

Introduction

In an optimising compilation process for instruction level parallelism, we may be faced to the opportunity of bounding the register pressure before instruction scheduling. This problem has been successfully studied in the literature (see our previous work on SIRA and register saturation).

An open problem arises for all strategies handling registers before instruction scheduling. Indeed, when we have a target processor with architecturally visible delays to access registers (such as in VLIW, DSP, EPIC and transport triggered architectures), the model of register requirement offers more opportunities to reduce the register pressure than in a regular sequential/superscalar processor. Such architectures are also called NUAL (non-unit-assumed-latencies).

Unfortunately, the opportunities offered by NUAL architectures are not fully or optimally exploited in existing register allocators. Two main reasons:

1. The exploitation of register access delay means the usage of non-positive edges latencies in data dependence graphs (DDG). As far as we know, current instruction schedulers do not exploit yet these sort of edges, and consider them as positive edges latencies. Furthermore, most of the theoretical results of scheduling in general deal with non-negative edges latencies.

2. If the register constraints are handled before instruction scheduling, an open problem arises regarding the possible creation of circuits with non-positive distances.

This reports studies the latter point. We show that:

- The circuits with non-positive distances define non-lexicographic positive DDG, which may not have a solution for instruction scheduling under resource constraints.
- Circuits with non-positive distances are not rare in practice, so the problem is not marginal.
- We show how to avoid the creation of non-positive circuits with two strategies: a reactive strategy (tolerate the problem then fix it if detected) and a proactive strategy (prevent the problem)

While the problem of non-positive circuits may arise in theory for any register optimisation method performing before SWP, this report shows how to avoid the problem in the SIRA framework: this is because, as far as we know, the SIRA framework is the only existing method that handles registers constraints before SWP while being sensitive to the increase of the initiation interval (II).

Our report is organised as follows. Chapter 2 defines the background of our work: in addition to the notations used in this report, we make a recall on the SIRA framework and SIRALINA heuristic (for self-containing). If the reader is familiar with our previous research results on SIRA and SIRALINA, he can skip this part and can go directly to Section 2.6. Section 2.6 presents a formal description of the problem solved in this report, as well as concrete examples to understand the intuition. Our reactive and proactive strategies are defined in Chapter 3: while the reactive strategy is easy to understand, the proactive strategy needs some formal understanding of the underlying scheduling problem. We study two separate heuristics included inside an iterative algorithm. Our heuristics are implemented and distributed inside the `SIRAlib` C-library: full experimental study of their efficiency on a large set of applications (FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006) is presented in Chapter 4. Then we conclude. Note that some additional experiments are inserted in the appendix: these extra experiments

are complementary to those presented in Chapter 4, but do not change the conclusions based on our empirical observations.

Chapter 2

Background

In this chapter, we recall briefly the SIRA framework[10] and SIRALINA heuristic [5, 3]. If the reader is already aware about these previous results, we invite him to skip the following sections and start the study from Section 2.6. Let start by defining some usual notations in our framework.

2.1 Mathematical notations and definitions

In this section, we introduce mathematical notations and definitions that are used afterwards.

A (directed) (multi)graph is a triple (V, E, ϕ) where V is a set of *vertices*, E is a set of *edges* and $\phi : E \to V \times V$. If $e \in E$ and $\phi(e) = (u, v)$, we define $src(e) = u$ and $tgt(e) = v$. We also use the term *node* instead of vertice.

In the sequel, we omit the ϕ component when manipulating graphs. Hence, a graph is just a pair (V, E) and we assume to have two functions $src : E \to V$ and $tgt : E \to V$ that define the *endpoints* of any edge $e \in E$.

By abuse of notation, we sometimes write $e = (u, v)$ when $src(e) = u$ and $tgt(e) = v$.

Given a graph $G = (V, E)$ we define:

- $In_G(u) = \{e \in E \mid tgt(e) = u\}$ the *input edges* of a vertex $u \in V$ in the graph G.
  $u \in V$ is a *source iff* $In_G(u) = \emptyset$.
- $Out_G(u) = \{e \in E \mid src(e) = u\}$ the *output edges* of a vertex $u \in V$ in the graph G.
  $u \in V$ is a *sink iff* $Out_G(u) = \emptyset$.
- A path in G is a sequence of edges $e_1, \ldots, e_n$ where $tgt(e_i) = src(e_{i+1})$ for $1 \leq i < n$ and $e_i \in E$ for $1 \leq i \leq n$.
- G is *acyclic iff* there are no path $e_1, \ldots, e_n$ in G such that $tgt(e_n) = src(e_1)$. Otherwise G is cyclic.
  Note that an acyclic graph has at least one source and one sink.
- A *cycle* in G is a directed path $e_1, \ldots, e_n$ in G such that $tgt(e_n) = src(e_1)$. We note $\mathcal{C}(G)$ the set of all the cycles in G. Hence G is acyclic *iff* $\mathcal{C}(G) = \emptyset$.

By abuse of notation, when G is clear from the context, we sometimes omit to precise it in some notations.

2.2 DDG model

A *data dependency graph* (DDG) is a directed graph $G = (V, E)$ where V is a set of vertices (instructions), E is a set of edges (data dependencies and serial constraints). Each edge $e \in E$ is labelled by a pair of values $(\delta(e), \lambda(e))$. $\delta : E \to \mathbb{Z}$ defines the latency of edges and $\lambda : E \to \mathbb{Z}$ defines the distance in terms of number of iterations. By abuse of notation, we sometimes write G for the underlying graph (V, E).

A DDG is said *lexicographic positive iff* all its cycles have positive distance, i.e. for any $c \in \mathcal{C}(G)$, $\lambda(c) = \sum_{e \in c} \lambda(e) > 0$. In other words, the characteristics $\forall c \in \mathcal{C}(G)$, $\lambda(c) > 0$ guarantees that the fully unrolled DDG is a DAG (there exists a topological sort between all the operation instances of the loop). In the sequel, we assume that the initial DDG is lexicographic positive, since it has been computed from an initial sequential code.

If cycle $c \in \mathcal{C}(G)$ with $\lambda(c) = \sum_{e \in c} \lambda(e) \leq 0$ exists, we simply call a *non-positive* cycle.

A *software pipelining* (SWP) is defined by a periodic schedule function $\sigma : V \to \mathbb{Z}$ and an *initiation interval* II, also called a period. Operation u of k^{th} iteration is scheduled at time $\sigma(u) + k \times \text{II}$. The schedule function is valid *iff* it satisfies the periodic precedence constraints

$$\forall e \in E : \sigma(src(e)) + \delta(e) \leq \sigma(tgt(e)) + \lambda(e) \times \text{II}$$

If G is cyclic, a necessary condition for a valid schedule to exist is that

$$\text{II} \geq \max_{c \in \mathcal{C}(G)} \frac{\sum_{e \in c} \delta(e)}{\sum_{e \in c} \lambda(e)} = \text{MII}$$

MII is called the *minimum initiation interval* defined by data dependences.

If G is acyclic, we define MII = 1 and not MII = 0. This is because no code generation is possible with MII = 0 (infinite parallelism).

2.3 Processor model

The modelled processor may have several register types: we note $\mathcal{T}$ the set of available register types.

An instruction which stores a value in a register of type $t \in \mathcal{T}$ is simply said to be a *value* of type t. Note that an instruction might be a value of several types simultaneously. However, our processor model does not support operations that store two or more values in registers of a given type t.

For a considered register type $t \in \mathcal{T}$, we note $V^{R,t}$ the set of statements $u \in V$ that are values of type t. We write u^t to define the value of type t created by the instruction u. Concerning the set of edges E, we distinguish between *flow* edges of type t —written $E^{R,t}$— and *serial* edges. A flow edge $e = (u, v)$ of type t represents the producer-consumer relationship between the two statements u and v. Serial edges are all other precedence constraints that are not flow edges of type t.

The set of *consumers* of a value $u \in V^{R,t}$ is

$$Cons^t(u) = \{tgt(e) \mid e \in E^{R,t} \wedge src(e) = u\}$$

NUAL and UAL semantics

Processor architectures can be decomposed into many families. One of the used classifications is related to the ISA code semantics [8]:

UAL code semantics : These processors have Unit-Assumed-Latencies at the architectural level. Sequential and superscalar processors belong to this family. In UAL semantic, the assembly code has a sequential semantic, even if the micro-architectural implementation executes instructions of longer latencies, in parallel, out of order or with speculation. The compiler instruction scheduler can always generate a valid code if it considers that all operations have a unit latency (even if such code may not be efficient).

NUAL code semantics : These processors have Non-Unit-Assumed-Latencies at the architectural level. VLIW, EPIC and some DSP processors belong to this family. In this sort of processors, the hardware pipeline steps (latencies, structural hazards, resource conflicts) are visible at the architectural level. Consequently, the compiler has to know about the instructions latencies, and sometimes with the underlying micro-architecture. The compiler instruction scheduler has to take care of these latencies to generate a correct code that does not violate data dependences.

Our processor model considers both UAL and NUAL semantics. Given a type $t \in \mathcal{T}$, we model possible delays when reading or writing registers of type t with two delay functions $\delta_{r,t}$ and $\delta_{w,t}$: these delay functions model NUAL semantics. Thus, the read cycle of u from a register of type t is $\sigma(u)+\delta_{r,t}(u)$ and the write cycle into a register of type t is $\sigma(u) + \delta_{w,t}(u)$. For UAL semantics, these delays are not visible and we have $\delta_{w,t} = \delta_{r,t} = 0$

The next section recalls the SIRA framework that allows to bound the register requirement before any SWP process.

2.4 SIRA: Schedule Independent Register Allocation

As argued in [10], satisfying the register constraints before instruction scheduling enjoys several benefits, especially regarding spill code elimination without hurting instruction level parallelism. A theoretical framework has thus been presented in [10] to address this problem, that we briefly recall in the following.

2.4.1 SIRA and Reuse Graphs

A simple way to explain and recall the concept of SIRA (Schedule Independent Register Allocation) is to provide an example. All the theory has already been presented in [10]. Figure 2.1(a) provides an initial DDG with two register types t_1 and t_2. Statements producing results of type t_1 are in dashed circles, and those of type t_2 are in bold circles. Statement u_1 writes two results of distinct types. Flow dependence through registers of type t_1 are in dashed edges, and those of type t_2 are in bold edges.

As an example, $Cons(u_2^{t_2}) = \{u_1, u_4\}$ and $Cons(u_3^{t_1}) = \{u_4\}$. Each edge e in the DDG is labelled with the pair of values $(\delta(e), \lambda(e))$. In this simple example, we assume that the delay of accessing registers is zero ($\delta_{w,t} = \delta_{r,t} = 0$). Now, the question is how to compute a periodic register allocation for the loop in Figure 2.1(a) without increasing the critical cycle if possible.

As formally studied in [10], periodic register constraints are modelled thanks to *reuse graphs*. We associate a reuse graph $G^{\text{reuse},t}$ to each register type t, see Figure 2.1(b). The reuse graph has to be computed by the SIRA framework, Figure 2.1(b) is one of the examples that SIRA may produce. Note that the reuse graph is not unique, other valid reuse graphs may exist.

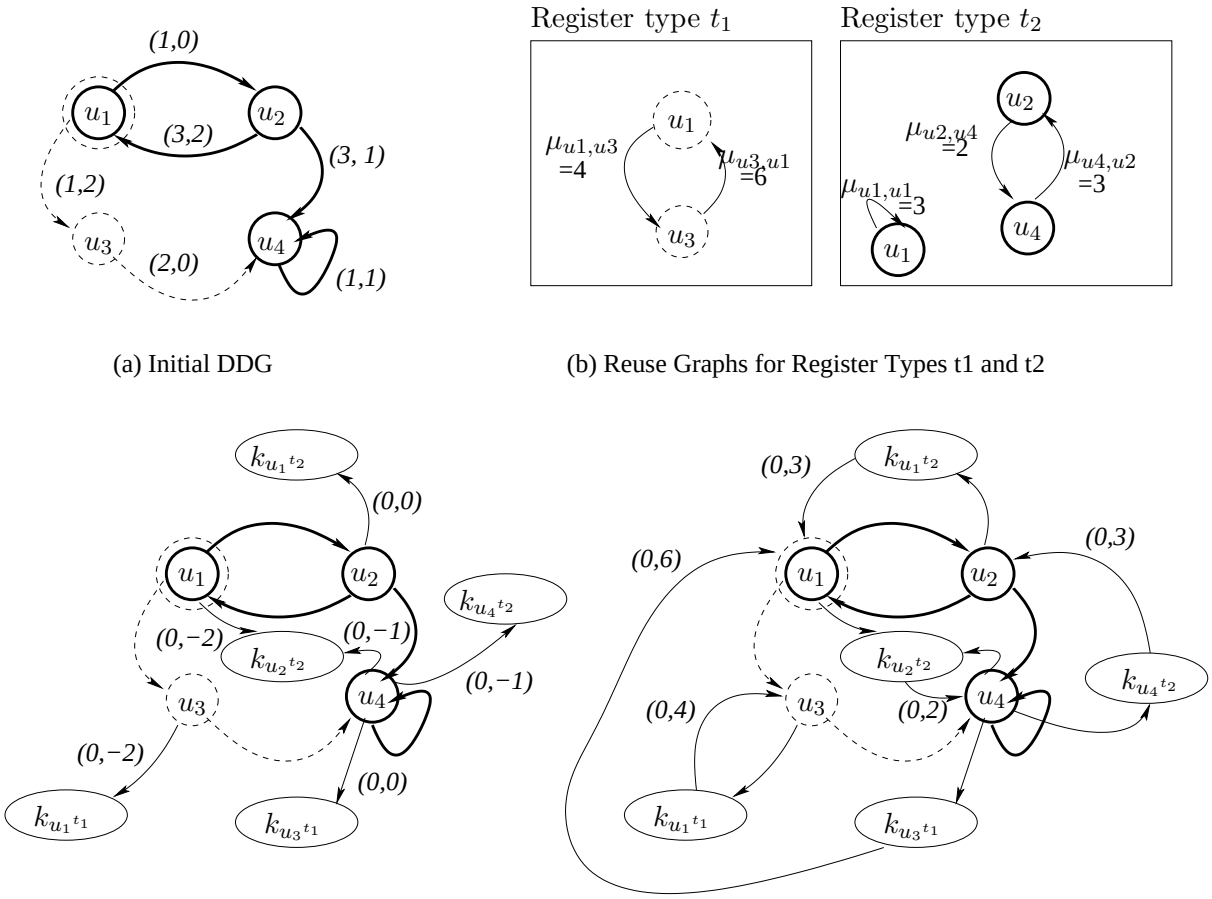
A reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ contains $V^{R,t}$, *i.e.*, only the nodes writing inside registers of type t. These nodes are connected by *reuse edges*. For instance, in G^{reuse,t_2} of Figure 2.1(b), the set of reuse edges is $E^{\text{reuse},t_2} = \{(u_2, u_4), (u_4, u_2), (u_1, u_1)\}$. Also, $E^{\text{reuse},t_1} = \{(u_1, u_3), (u_3, u_1)\}$. Each reuse edge $e_r = (u, v)$ is labelled by an integral distance $\mu^t(e_r)$. The existence of a reuse edge $e_r = (u, v)$ of distance $\mu^t(e_r)$ means that the two operations $u(i)$ and $v(i + \mu^t(e_r))$ *share the same destination register* of type t. Hence, reuse graphs allows to completely define a periodic register allocation for a given loop. In the example of Figure 2.1(b), we have in G^{reuse,t_2} $\mu^{t_2}((u_2, u_4)) = 2$ and $\mu^{t_2}((u_4, u_2)) = 3$.

In order to be valid, reuse graphs should satisfy two main constraints [10]: 1) They should describe a bijection between the nodes; that is, they must be composed of elementary and disjoint cycles. 2) The *associated DDG* should be schedulable, *i.e.*, it has at least one valid SWP.

Now, let us describe what we mean by the DDG *associated with* a reuse graph. Once a reuse graph is fixed before SWP, say the reuse graphs of types t_1 and t_2 in Figure 2.1(b), the register constraints create new periodic scheduling constraints between loop statements. These scheduling constraints result from the anti-dependencies created by register reuse. Since each reuse arc (u, v) in the reuse graph $G^{\text{reuse},t}$ describes a register sharing between $u(i)$ and $v(i + \mu^t((u, v)))$, we must guarantee that $v(i + \mu^t((u, v))$ writes inside the same register after the execution of all the consumers of $u^t(i)$. That is, we should guarantee that $v(i + \mu^t((u, v)))$ writes its result after the killing date of $u^t(i)$. If the loop is already scheduled, the killing date is known. However, if the loop is not already scheduled, then the killing date is not known and hence we should be able to guarantee the validity of periodic register allocation for all possible SWP schedules.

Guaranteeing precedence relationship between lifetime intervals for any subsequent SWP is done by creating *the associated DDG* with the reuse graph. This DDG is an extension of the initial one in two steps:

1. **Killing nodes:** First, we introduce dummy nodes representing the killing dates of all values[2]. For each value $u \in V^{R,t}$, we introduce a node k_{u^t} which represents the killing date of u^t. The



(a) Initial DDG

(b) Reuse Graphs for Register Types t1 and t2

(c) DDG with Killing Nodes

(d) Preconditioned DDG after Applying SIRA

Figure 2.1: Example for SIRA and Reuse Graphs

killing node k_{u^t} must always be scheduled after all u^t's consumers, so we add edges of the form $e = (v, k_{u^t})$ where $v \in Cons^t(u)$. If a value u^t has no consumer (not read inside the loop), it means that the node can be killed just after the creation of its result. Figure 2.1(c) illustrates the DDG after adding all the killing nodes for all register types. For each added edge $e = (v, k_{u^t})$, we set its latency to $\delta(e) = \delta_{r,t}(v)$ and its distance to $-\lambda$, where λ is the distance of the flow dependence edge $(u, v) \in E^{R,t}$. As explained in [10], this negative distance is a mathematical convention, it simplifies our mathematical formula and does not influence the fundamental results of reuse graphs. Formally, if $u \in V^{R,t}$ is a node writing a value of type $t \in \mathcal{T}$, then we note k_{u^t} the killer node of type t of the value u^t. The set of killing nodes of type t is noted $V^{k,t}$. For each type $t \in \mathcal{T}$, we note $E^{k,t}$ the set of edges defining the precedence constraints between $V^{R,t}$ nodes and the killer nodes:

$$E^{k,t} = \{e = (v, k_{u^t}) \mid u \in V^{R,t} \wedge v \in Cons^t(u) \wedge \delta(e) = \delta_{r,t}(v)\}$$
$$\cup \{(u, k_{u^t}) \mid u \in V^{R,t} \wedge Cons^t(u) = \emptyset \wedge \delta(e) = 1\}$$

For instance, in Figure 2.1(b), we have $V^{k,t_2} = \{k_{u_1{}^{t_2}}, k_{u_2{}^{t_2}}, k_{u_4{}^{t_2}}\}$, and we have $E^{k,t_2} = \{(u_2, k_{u_1{}^{t_2}}), (u_1, k_{u_2{}^{t_2}}), (u_4, k_{u_2{}^{t_2}}), (u_4, k_{u_4{}^{t_2}})\}$.

If we note $K = \bigcup_{t \in \mathcal{T}} V^{k,t}$ and $E^k = \bigcup_{t \in \mathcal{T}} E^{k,t}$, then the DDG with killing nodes is defined by $(V \cup K, E \cup E^k)$.

2. **Anti-dependence edges:** Second, we introduce new anti-dependence edges implied by periodic register constraints. For each reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$, we add an edge $e'_r = (k_{u^t}, v)$

representing an *anti-dependence* in the associated DDG. We say that the anti-dependence $e'_r = (k_{u^t}, v)$ in the DDG G is associated with the reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$. We write $\Phi(e_r) = e'_r$ and $\Phi^{-1}(e'_r) = e_r$.

The added anti-dependence edge $e'_r = (k_{u^t}, v)$ has a distance equal to the reuse distance $\lambda(e'_r) = \mu^t(e_r)$, and a latency equal to:

- $\delta(e'_r) = -\delta_{w,t}(v)$ if the processor has NUAL semantics.
- $\delta(e'_r) = 1$ if the processor has UAL semantics. Note that we can still assume a latency $\delta(e'_r) = \delta_{w,t} - \delta_{r,t} = 0$, since the instruction scheduler will generate a sequential code, so this zero edge imposes to schedule k_{u^t} before v.

Figure 2.1(d) illustrates the DDG associated to the two reuse graphs of Figure 2.1(b). Periodic register constraints with multiple register types are satisfied conjointly on the same DDG even if each register type has its own reuse graph. The reader may notice that the critical cycle of the DDG in Figure 2.1(a) and (c) are the same and equal to MII $= \frac{4}{2} = 2$ (a critical cycle is (u_1, u_2)). The set of added anti-dependence edges of type t is noted $E^{\mu,t}$

$$E^{\mu,t} = \{e = (k^t_u, v) \mid e_r = (u, v) \in E^{\text{reuse},t} \wedge \Phi(e_r) = e\}$$

In Figure 2.1(d), $E^{\mu,t_1} = \{(k_{u_1{}^{t_1}}, u_3), (k_{u_3{}^{t_1}}, u1)\}$ and $E^{\mu,t_2} = \{(k_{u_1{}^{t_2}}, u_1), (k_{u_2{}^{t_2}}, u_4), (k_{u_4{}^{t_2}}, u_2)\}$.

If we note $E^{\mu} = \bigcup_{t\in\mathcal{T}} E^{\mu,t}$, then the DDG G' (with killing nodes) associated with the reuse graphs $\left(V^{R,t}, E^{\text{reuse},t}\right)_{t\in\mathcal{T}}$ is defined by $G' = (\mathcal{V} = V \cup K,\ \mathcal{E} = E \cup E^k \cup E^{\mu})$.

As can be seen, computing a reuse graph of a register type t implies the creation of new edges with μ^t distances. We proved in [9] that if a reuse graph $G^{\text{reuse},t}$ is valid, then it describes a periodic register allocation with exactly $\sum_{e_r\in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t. That is, any SWP schedule cannot require more than $\sum_{e_r\in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t, and this upper-bound is reachable.

Now the SIRA problem is to compute a valid reuse graph with minimal $\sum_{e_r\in E^{\text{reuse},t}} \mu^t(e_r)$, without increasing the critical cycle if possible. Or, instead of minimising the register requirement, SIRA may simply look for a solution such that $\sum_{e_r\in E^{\text{reuse},t}} \mu^t(e_r) \leq \mathcal{R}^t$, where $\mathcal{R}^t$ is the number of available registers of type t. We may propose many exact method models (the problem has been proved NP-complete in [10]) or heuristics based on the SIRA framework. The following section presents SIRALINA, an efficient two steps heuristic.

2.5 SIRALINA Heuristic

Computing a valid reuse graph for a fixed period II that minimises $\sum_{e_r\in E^{\text{reuse},t}} \mu^t(e_r)$ is NP-complete [9]. SIRALINA [3, 5] is an efficient heuristic for this problem for all register types conjointly. In order to balance between the importance of each involved register type, we assume to have a weight $\alpha_t \in \mathbb{R}$ attributed to each type $t \in \mathcal{T}$. This weight may be set to 1 if all register types have the same importance.

SIRALINA decomposes the SIRA linear optimisation into two polynomial steps summarised as follows (here, the period II is fixed):

1. Step 1 (scheduling problem): Determine minimal reuse distances for all pairs of values (i.e. compute, for each type t, a function $\hat{\mu}^t : V^{R,t} \times V^{R,t} \to \mathbb{Z}$);
2. Step 2 (linear assignment problem): Determine a bijection $E^{\text{reuse},t} : V^{R,t} \to V^{R,t}$ that minimises $\sum_{(u,v)\in V^{R,t}\times V^{R,t}} \hat{\mu}^t(u, v)$ for each t.

These two steps allows the construction of a reuse graph for a period II. Then $G' = (\mathcal{V}, \mathcal{E})$ the associated DDG is constructed as explained previously: $\mathcal{V} = V \cup K$ and $\mathcal{E} = E \cup E^k \cup E^{\mu}$. The two following sections details each of the two above steps.

2.5.1 Step 1: the scheduling problem for a fixed II

The scheduling problem is to guarantee the existence of a SWP schedule for the associated DDG. The problem is formulated as an integer linear problem with totally unimodular constraints matrix. In addition, it aims at determining minimal reuse distances for all pairs of values.

Integer variables of the linear problem
For any $u \in \mathcal{V}$, define a variable $\sigma_u \in \mathbb{Z}$ representing a scheduling date.

Linear program formulation
The scheduling problem is expressed as follows:

$$\begin{cases} \text{minimise} & \displaystyle\sum_{t \in \mathcal{T}} \alpha_t \left(\sum_{u \in V^{R,t}} \sigma_{k_u^t} - \sum_{u \in V^{R,t}} \sigma_u \right) \\ \text{subject to} & \\ & \forall e \in E \cup E^k, \sigma_{tgt(e)} - \sigma_{src(e)} \geq \delta(e) - \mathrm{II} \times \lambda(e) \end{cases}$$

The constraints matrix of this integer linear program is an incidence matrix of the DDG G (with killing nodes), consequently it is totally unimodular. Hence it can be solved with a polynomial algorithm.

Let ${\sigma_u}^*$ and ${\sigma_{k_u^t}}^*$ be the values of the variables of the optimal solution of the above scheduling problem. The minimal reuse distance function is then defined as follows for all pairs of values (u, v).

$$\hat{\mu^*}^t(u,v) = \left\lceil \frac{{\sigma_{k_u^t}}^* - \delta_{w,t}(v) - {\sigma_v}^*}{\mathrm{II}} \right\rceil$$

2.5.2 Step 2: the linear assignment problem

The linear assignment problem for a register type t is to find a bijection $\theta^t : V^{R,t} \rightarrow V^{R,t}$ such that $\sum_{u \in V^{R,t}} \hat{\mu^*}^t(u, \theta^t(u))$ is minimal. It can be solved in polynomial time complexity with the so-called Hungarian algorithm[6]. Such an optimal bijection θ^t defines a set of reuse edges $E^{\mathrm{reuse},t}$ as follows.

$$E^{\mathrm{reuse},t} = \{e_r = (u, \theta^t(u)) \mid u \in V^{R,t} \wedge \mu^t(e_r) = \hat{\mu^*}^t(u, \theta^t(u))\}$$

After executing the two steps of SIRALINA, the valid reuse graphs (one for each register type t) are defined above, and the initial DDG G is extended by adding edges as explained in Section 2.4.1. The next section describes the problem of cycles with non-positive distances that may introduced in this extended DDG (associated DDG to the reuse graphs).

2.6 Problem Description of Non Lexicographic-Positive Cycles

A cycle C is said lexicographic-positive iff $\lambda(C) > 0$, while $\lambda(C)$ is a notation for $\sum_{e \in C} \lambda(e)$. A data dependence graph (DDG) is said lexicographic-positive iff all its cycles are so too. A DDG is said schedulable iff there exists a valid SWP, *i.e.*, a SWP satisfying all its cyclic precedence constraints, not necessarily satisfying other constraints such as resources or registers. An data dependence graph computed from a sequential program is always lexicographic positive, it is an inherent characteristic of imperative sequential languages. When a DDG is lexicographic-positive, there is a guarantee that a schedule exists for it (at least the initial sequential schedule).

Since SIRA is applied before instruction scheduling, it modifies the DDG under the condition that it remains schedulable. If the target architecture has a UAL code semantics (sequential code), then the introduced edges by any SIRA method (such as SIRALINA) has unit-assumed latencies, and the DDG remains lexicographic positive. If the target architecture has explicit architectural delays in accessing registers (NUAL code semantics), then the introduced edges by SIRALINA are of the form $e' = (k_{u^t}, v)$ with latencies $\delta(e') = -\delta_{w,t}(v)$. Such latencies are non-positive.

If an edge latency is non-positive, this does not create specific problem for cyclic scheduling in theory, unless if the latency of a cycle is negative too. The following lemma proves that if $\delta(C) < 0$, then the DDG may not be lexicographic positive.

Lemma 1. *Let G be a schedulable DDG with SWP. Let C be an arbitrary cycle in G. Then the following implications are true:*

1. $\delta(C) \geq 0 \Longrightarrow \lambda(C) \geq 0$.
2. $\delta(C) \leq 0 \Longrightarrow \lambda(C)$ *may be non-positive.*

Proof. Since the DDG is schedulable, then there exists a valid SWP with $II > 0$. It is well known that $\forall C$ a cycle $: II \times \lambda(C) \geq \delta(C)$, hence $\lambda(C) \geq \frac{\delta(C)}{II}$.

1. $II > 0 \wedge \delta(C) \geq 0 \Longrightarrow \lambda(C) \geq 0$.
2. $II > 0 \wedge \delta(C) \leq 0 \Longrightarrow \lambda(C) \geq x$, where $x = \frac{\delta(C)}{II} \leq 0$.

□

The previous lemma proves that inserting negative edges inside a DDG can generate cycles with $\lambda(C) \leq 0$. So, what is the problem with such cycles? Indeed, the answer comes from the cyclic scheduling theory. Given a cyclic DDG, let C^+ be the set of cycles with $\lambda(C) > 0$, le C^- be the set of cycles with $\lambda(C) < 0$, and let C^0 be the set of cycles with $\lambda(C) = 0$. Then the following inequality is true[1]:

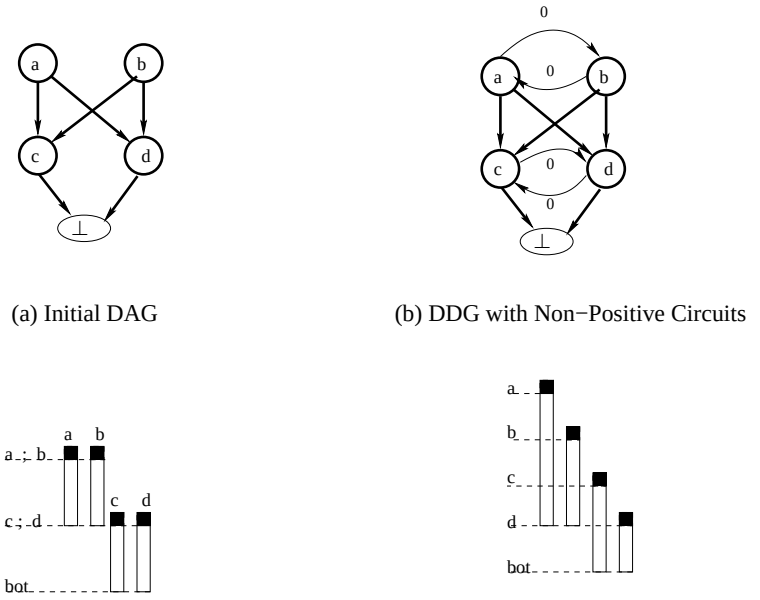
$$\max_{C \in C^+} \frac{\delta(C)}{\lambda(C)} \leq II \leq \min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$$

In other words, the existence of cycles inside C^- imposes hard real time constraints on the value of II. Such constraints can be satisfied with cyclic scheduling if we consider only precedence constraints[1]. However, if we add resource constraints (as will be carried out during the subsequent instruction scheduling pass), then the DDG may not be schedulable. Simply it may be possible that the conflicts on the resources may not allow to have an II lower than $\min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$.

When a circuit $C \in C^0$ exists, this means that there is a precedence relationship between the statements belonging to the same iteration: that is, the loop body is no longer an acyclic graph as in the initial DDG.

2.6.1 Simple Examples with DAG and with Cyclic DDG

In order to understand the intuition behind the problem lexicographic-negative circuits, let consider an acyclic scheduling problem on a DAG. So the scheduling problem considers only latencies $\delta(C)$, no recurrent dependences through circuits. Let consider the DAG of Figure 2.2 (a). All bold nodes write a value inside a register, the data flow edges are in bold and have a latency equal to 1. If we do not



(a) Initial DAG

(b) DDG with Non−Positive Circuits

(c) Parallel Code (requires 2 registers)

(d) Sequential Code (requires 3 registers)

Figure 2.2: Example of an Acyclic Schedule with Non-Positive Circuits

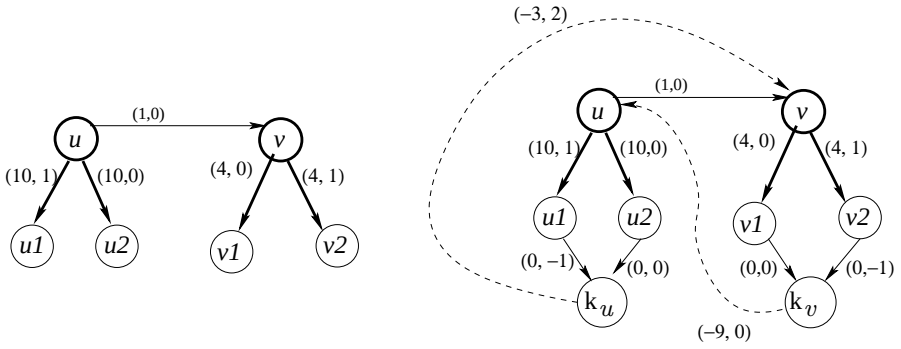
consider any resource constraints, both the acyclic schedules in Figures 2.2 (c) and (d) are valid, since they satisfy the precedence constraints of the DAG.

Now, if we consider register constraints, not all schedules have the same register requirement. Figures 2.2 (c) and (d) contain the lifetime intervals of the 4 values of the DAG. While the parallel schedule in Figures 2.2 (c) requires 2 registers (two values simultaneously alive), the schedule of Figure 2.2 (d) requires 3 registers. A DDG that guarantees that all valid acyclic schedules require 2 registers is presented in Figure 2.2 (b): this DDG is no longer a DAG. It contains two circuits with a latency equal to zero. This DDG is schedulable with a valid acyclic schedule: Figures 2.2 (c)) defines a valid schedule if we do not consider resource constraints. If we consider resource constraints, the DDG may not be schedulable. Simply consider that all the instructions $\{a, b, b, d\}$ use the same resource, consequently the parallel schedule of Figure 2.2 (c) is not possible because it violates a resource constraint. Also, the sequential schedule of Figure 2.2 (d) is not valid because it violates the precedence constraints of the DDG in Figure 2.2 (b).

The simple example of Figure 2.2 gives us an intuition that non-positive circuits do not prevent a DDG from being schedulable, but may be so in presence of resources constraints.

Now, we can construct an example for cyclic scheduling. Consider Figure 2.3, where flow dependences are in bold edges and loop instructions writing into registers are in bold circles. We assume that the target processor have visible delays in accessing registers. The initial DDG shown in Figure 2.3 (a) is acyclic, but represents a cyclic scheduling problem:. The latency of u is equal to 10, and the latency of v is equal to 4. We assume that $\delta_{r,t} = 0$ for all instructions, but $\delta_{w,t}(u) = 9$ and $\delta_{w,t}(v) = 3$. There exists a dependence path from u to v with a distance equal to zero (the path is in the loop body). After applying a SIRA method on this DDG to bound the register requirement, the DDG can be modified as shown in Figure 2.3 (b). Here the SIRA method assigns the same destination register to $u(i)$ and $v(i)$ (statements in the same loop iteration). This creates an anti-dependence from $v(i)$'s killer to $u(i)$. Since the latency of the inserted edge (k_v, u) is negative (-9) and the latency of the path $u \rightsquigarrow k_v$ is 5, the circuit (v, k_v, u, v) has a distance equal to zero. This DDG is schedulable but is not lexicographic-positive.

As a compiler construction strategy, we must guarantee that the schedulable DDG produced after applying SIRA is always lexicographic positive. Otherwise, there is no guarantee that the subsequent SWP pass would find a solution, and the code generation may fail. This problem is studied and solved in the next chapter.



(a) Initial Loop DDG (b) Modified DDG

Figure 2.3: Example of a Loop DDG with Non-Positive Circuits

Chapter 3

Eliminating Non-Positive Cycles with SIRALINA

As mentioned previously, we need to ensure that the associated DDG computed by SIRALINA is lexicographic positive.

We have also noted that if the processor has a UAL semantics then it is guaranteed that any associated DDG found by SIRALINA is lexicographic positive. This is because the UAL semantic is used to model sequential processors, all inserted anti-dependences edges have latency equal to 1. Since all the edges in the associated DDG have positive latencies, and since the associated DDG is schedulable by SWP (guaranteed by SIRALINA), then the DDG is necessarily lexicographic positive.

Hence, a naive strategy is to always consider UAL semantics. That is, we do not exploit the access delays to registers. This solution works in practice but the register requirement model is not optimal, since it does not exploit NUAL code semantics. Consequently, the computed register requirement is not well optimised.

A more clever, yet naive, way to ensure that any associated DDG computed by SIRALINA is lexicographic positive is to have a *reactive* strategy. It tolerates the problem as follows:

1. Solve SIRALINA considering NUAL semantics.

2. Check whether the associated DDG is lexicographic positive and

   - if it is, then return the computed solution.
   - if it is not, then apply SIRALINA considering UAL semantics.

Considering a UAL semantic for SIRALINA on a processor that has a NUAL semantics cannot hurt: it just possibly implies a loss of optimality in either II or in the register requirement. The above method is optimistic (reactive) in the sense that it considers that non lexicographic DDG are rare in practice. This is not true in theory of course, but maybe the practice would highlight that the proportion of the problems producing DDG that must be *corrected* is low. In this case, it is in practice better to do not try to restrict SIRALINA, but to correct the solution afterwards if we detect the problem.

The question that thus arises naturally is the following: is it possible to devise a better method to ensure *a priori* that the associated DDG computed by SIRALINA are lexicographic positive while exploiting the benefit of NUAL semantics ? That is, we are willing to study a *proactive* strategy that prevents the problem.

In the sequel we describe two distinct proactive methods, similar in spirit, that address the problem of non-positive cycles. The first one is based on retiming ideas while the second one is based on simple results of graph theory.

3.1 Overview of the two proactive methods for eliminating non-positive cycles

The two methods presented afterwards are based on the same idea. Before describing the common scheme of the two methods, first recall that SIRALINA works as follows:

1. Firstly determine, for each type t, minimal reuse distances for all pairs of values of type t, i.e. compute a function $\hat{\mu^*}^t : V^{R,t} \times V^{R,t} \to \mathbb{Z}$.
2. Secondly, for each type t, determine a bijection of $\theta^t : V^{R,t} \to V^{R,t}$ that minimises $\sum_{(u,v)\in V^{R,t}\times V^{R,t}} \hat{\mu^*}^t(u,v)$. θ^t defines the set of reuse edges $E^{\text{reuse},t} = \{e_r = (u, \theta^t(u)) \mid \mu^t(e_r) = \hat{\mu^*}^t(u,v)\}$

Recall also that the number of required registers of type t is $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$.

Once a set of reuse edges is determined, the associated DDG is defined as explained in Section 2.4.1. Since we assume that the initial DDG is lexicographic positive, it is clear that if the associated DDG is not lexicographic positive, then any cycle of non-positive distance necessary contains at least one edge $(k_u^t, v) \in E^{\mu,t}$ associated to $(u,v) \in E^{\text{reuse},t}$.

Our idea is thus the following. Once a set of reuse edges is determined by the second step of SIRALINA, we increment the distance functions $\hat{\mu}^t$ so that the associated DDG to the current set of reuse edges does not contain any cycle of negative distance. Incrementing reuse distances is always a valid transformation if it does not violate the scheduling constraints. However, this transformations may ask to use more registers.

Indeed, observe that in the associated DDG, the added edges $e'_r = (k_u^t, v) \in E^{\mu,t}$, where $e_r = (u,v)$ is a reuse edge of type t, have a distance equal to $\lambda(e) = \hat{\mu}^t(u,v)$, and that the distances of the other edges are entirely determined by the initial DDG and are not subject to changes. By modifying $\hat{\mu}^t(u,v)$, it may happen that a better set of reuse edges (i.e. a better solution to the assignment problem) exists, since the distance functions $\hat{\mu}^t$ may have changed. In this case, we may choose to backtrack our choice of reuse edges and redo the entire SIRALINA process. This defines an iterative process. We may decide to stop after a certain number of iterations since it is not clear that the process terminates otherwise.

Our modified SIRALINA heuristic is thus given by Algorithm 1. At each iteration i of the algorithm, it computes new reuse distances $\hat{\mu}^t_{(i)}$ and new reuse edges $E^{\text{reuse},t}_{(i)}$, based on the previous reuse distances $\hat{\mu}^t_{(i-1)}$ and previous reuse edges $E^{\text{reuse},t}_{(i-1)}$. This algorithm is parametrised by two functions:

- $LinearAssignment(G, \hat{\mu}^t)$ computes a bijection $\theta^t : V^{R,t} \times V^{R,t}$ that minimises $\sum_{(u,v)\in V^{R,t}\times V^{R,t}} \hat{\mu}^t(u,v)$. In other words, it solves the linear assignment problem and is typically implemented by the Hungarian algorithm, as done in the second step of SIRALINA. The result of this function is the set of reuse edges $E^{\text{reuse},t}$.
- $UpdateReuseDistances(G, (\hat{\mu}^t)_{t\in\mathcal{T}}, (E^{\text{reuse},t})_{t\in\mathcal{T}})$ computes new distance functions $(\hat{\mu}^t)_{t\in\mathcal{T}}$ such that the associated DDG w.r.t.$(\hat{\mu}^t)_{t\in\mathcal{T}}$ and $(E^{\text{reuse},t})_{t\in\mathcal{T}}$ is lexicographic positive.

The body of the repeat-until loop is executed at most $n+1$ times and is interrupted when a fix-point is reached, *i.e.* when the set of reuse edges stabilises from one iteration to another ($E^{\text{reuse},t}_{(i)} = E^{\text{reuse},t}_{(i-1)}$). Since the body of algorithm loop is executed at least once, it is guaranteed that the associated DDG will be lexicographic positive.

The two following sections explains our two possible implementations of the function $UpdateReuseDistances$.

3.2 Iterative method based on circuit retiming

The first method is based on circuit retiming idea [7]. By using circuit retiming, we proved the following result [9, 10]: the associated DDG is lexicographic positive *if and only if* there exists $|\mathcal{E}|$ retiming functions $r_e : V \to \mathbb{Z}$ for $e \in \mathcal{E}$ such that

$$\begin{cases} \forall e \in \mathcal{E}, \forall e' \neq e, & r_e(tgt(e')) - r_e(src(e')) + \lambda(e') \geq 0 \\ \forall e \in \mathcal{E}, & r_e(tgt(e)) - r_e(src(e)) + \lambda(e) \geq 1 \end{cases}$$

Algorithm 1 Iterative SIRALINA to produce associated lexicographic-positive DDG

function ITERATIVESIRALINA(G, n)
▷ G is the initial DDG
▷ $n \geq 0$ is a bound on the maximal number of iterations
$(\hat{\mu}^t_{(0)})_{t\in\mathcal{T}} \leftarrow (\hat{\mu^*}^t)_{t\in\mathcal{T}}$ ▷ Compute initial distance functions by solving the scheduling problem
for $t \in \mathcal{T}$ **do**
$E^{\text{reuse},t}_{(0)} \leftarrow LinearAssignment(G, \hat{\mu}^t_{(0)})$
end for
$i \leftarrow 0$
repeat
$i \leftarrow i + 1$
$(\hat{\mu}^t_{(i)})_{t\in\mathcal{T}} \leftarrow UpdateReuseDistances(G, (\hat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}, (E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}})$
for $t \in \mathcal{T}$ **do**
$E^{\text{reuse},t}_{(i)} \leftarrow LinearAssignment(G, \hat{\mu}^t_{(i)})$
end for
if $E^{\text{reuse},t}_{(i)} = E^{\text{reuse},t}_{(i-1)}$ for every $t \in \mathcal{T}$ **then**
break ▷ A fix-point has been reached
end if
until $i > n$
return $(\hat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ and $(E^{\text{reuse},t}_{(i)})_{t\in\mathcal{T}}$
end function

Intuitively, the existence of a retiming function r_e for a given edge e means that there are no cycle of non-positive distance that contains e. The system defined above ensures thus simply that for any edge e, there exists a retiming function r_e, which thus means that there are no cycle of non-positive distance that contains e for any e.

To find those retiming functions, we define a mixed-integer linear program. For any edge $e \in \mathcal{E}$ and vertex u, we define an *integer* variable $r_{e,u}$.

For each anti-dependence edge $e = (k^t_u, v) \in E^{\mu,t}$ corresponding to the reuse edge $e_r = (u, v)$, we define a variable $\gamma^t(u, v)$, so that the distance of e is $\lambda(e) = \mu^t(\Phi^{-1}(e)) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$. In iterative SIRALINA (Algorithm 1), $\hat{\mu}^t(u, v)$ have been already computed in a previous step, so we seek to update the new reuse distance by adding the increment $\gamma^t(u, v) \in \mathbb{Z}$ to the previous reuse distance $\hat{\mu}^t(u, v)$.

The above system of constraints is expressed as:

$$\begin{cases} \forall e \in \mathcal{E}, \forall e' \neq e, & r_{e,tgt(e')} - r_{e,src(e')} + \lambda(e') \geq 0 \\ \forall e \in \mathcal{E}, & r_{e,tgt(e)} - r_{e,src(e)} + \lambda(e) \geq 1 \end{cases}$$

Since the needed number of registers of type t is $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e)$, we seek to minimise $\sum_{e_r=(u,v)\in E^{\text{reuse},t}} \gamma^t(u, v)$ for any type t. We approximate this multi-objective by attributing a weight α_t to each type t and minimising $\sum_{t\in\mathcal{T}} \alpha_t \sum_{e_r=(u,v)\in E^{\text{reuse},t}} \gamma^t(u, v)$. The weight α_t can be fixed according to the relative importance of the register types, or it can simply be fixed as equal to 1 for all register types.

Moreover, in order to ensure the validity of the reuse graph, we must satisfy the scheduling constraints. That is, we require that $\hat{\mu}^t(u, v) + \gamma^t(u, v) \geq \hat{\mu^*}^t(u, v)$ for any $(u, v) \in V^{R,t} \times V^{R,t}$, where $\hat{\mu^*}^t(u, v)$ is the solution of the scheduling problem (computed in the first step of SIRALINA).

We thus finally have defined the following mixed-integer linear program, which contains $O(|\mathcal{E}| \times |\mathcal{V}|)$ variables and $O(|\mathcal{E}|^2)$ linear constraints. It is given on Figure 3.1.

Once a solution is found for the the system of Figure 3.1, we set the new reuse distance between of $e = (k^t_u, v) \in E^{\mu,t}$ as $\lambda(e) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$.

Since the number of registers is integer, we have to ensure that $\gamma^t(u, v)$ is integer. However, it is easy to see that it is sufficient to work with a relaxed version of the above problem where the type of $\gamma^t(u, v)$ is not constrained and then define the reuse distance of (u, v) as $\hat{\mu}^t(u, v) + \lceil \gamma^t(u, v) \rceil$. This relaxed version is shown on Figure 3.2. Remember that for edges $e = (k^t_u, v) \in E^{\mu,t}$, corresponding to a reuse edge $e_r = (u, v) \in E^{\text{reuse},t}$, we have $\lambda(e) = \mu^t(\Phi^{-1}(e)) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$, and $\hat{\mu}^t(u, v)$ has already been

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{subject to} & \\
\forall e \in \mathcal{E}, & r_{e,tgt(e)} - r_{e,src(e)} + \lambda(e) \geq 1 \\
\forall e \in \mathcal{E}, \forall e' \neq e \in \mathcal{E}, & r_{e,tgt(e')} - r_{e,src(e')} + \lambda(e') \geq 0 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \hat{\mu}^t(u,v) + \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) \\
\forall e \in \mathcal{E}, \forall u \in \mathcal{V}, & r_{e,u} \in \mathbb{Z} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \gamma^t(u,v) \in \mathbb{Z} \\
\text{where} & \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & \lambda(e) \stackrel{\text{def}}{=} \hat{\mu}^t(u,v) + \gamma^t(u,v)
\end{cases}
$$

Figure 3.1: Mixed-Integer linear program based on circuit retiming

computed in the previous iteration of Algorithm 1: $\hat{\mu}^t(u,v) = \hat{\mu}^t_{(i-1)}(u,v)$, $E^{\text{reuse},t} = E^{\text{reuse},t}_{(i-1)}$ and the set $E^{\mu,t} = \{\Phi(e_r) \mid e_r \in E^{\text{reuse},t}_{(i-1)}\}$. Thus, we give in Figure 3.3 a more explicit version of the linear program of Figure 3.2.

Hence our implementation of the function $UpdateReuseDistances(G, (\hat{\mu}^t)_{t\in\mathcal{T}}, (E^{\text{reuse},t})_{t\in\mathcal{T}})$ is given by Algorithm 2.

Algorithm 2 Heuristic based on circuit retiming

function UPDATEREUSEDISTANCES($(G, (\hat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}, (E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}})$)

Solve the mixed-integer linear program of Figure 3.3 to compute $(\gamma^t(u,v))$

return $(\hat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ where $\hat{\mu}^t_{(i)}(u,v) \stackrel{\text{def}}{=} \begin{cases} \hat{\mu}^t_{(i-1)}(u,v) + \lceil \gamma^t(u,v) \rceil & \text{if } (u,v) \in E^{\text{reuse},t}_{(i-1)} \\ \hat{\mu}^t_{(i-1)}(u,v) & \text{otherwise} \end{cases}$

end function

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{subject to} & \\
\forall e \in \mathcal{E}, & r_{e,tgt(e)} - r_{e,src(e)} + \lambda(e) \geq 1 \\
\forall e \in \mathcal{E}, \forall e' \neq e \in \mathcal{E}, & r_{e,tgt(e')} - r_{e,src(e')} + \lambda(e') \geq 0 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \hat{\mu}^t(u,v) + \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) \\
\forall e \in \mathcal{E}, \forall u \in \mathcal{V}, & r_{e,u} \in \mathbb{Z} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \gamma^t(u,v) \in \mathbb{R}, \\
\text{where} & \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & \lambda(e) \stackrel{\text{def}}{=} \hat{\mu}^t(u,v) + \gamma^t(u,v)
\end{cases}
$$

Figure 3.2: Relaxed linear program based on circuit retiming

Unfortunately, even the relaxed version in Figure 3.3 is in theory difficult to solve because it involves integer variables and it has not the shape of an *easy* integer linear problem, for which efficient procedures exist.

That is why we have worked for a second method, which is in theory easier to solve (and in practice too, as we will see later). The next section describes the SPE method.

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\\
\text{subject to:} & \\
\forall e \in E \cup E^k, & r_{e,tgt(e)} - r_{e,src(e)} + \lambda(e) \geq 1 \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & r_{e,tgt(e)} - r_{e,src(e)} + \gamma^t(u,v) \geq 1 - \hat{\mu}^t_{(i-1)}(u,v) \\
\forall e \in \mathcal{E}, \forall e' \neq e \in E \cup E^k, & r_{e,tgt(e')} - r_{e,src(e')} + \lambda(e') \geq 0 \\
\forall e \in \mathcal{E}, \forall t \in \mathcal{T}, \forall e' = (k_u^t, v) \neq e \in E^{\mu,t}, & r_{e,tgt(e')} - r_{e,src(e')} + \gamma^t(u,v) \geq -\hat{\mu}^t_{(i-1)}(u,v) \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) - \hat{\mu}^t_{(i-1)}(u,v) \\
\forall e \in \mathcal{E}, \forall u \in \mathcal{V}, & r_{e,u} \in \mathbb{Z} \\
\forall t \in \mathcal{T}, \forall e_r \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(e_r) \in \mathbb{R}
\end{cases}
$$

Figure 3.3: Relaxed linear program based on circuit retiming (full linear constraints)

3.3 Iterative method based on shortest paths equations (SPE)

Our second proactive method, named SPE, is based on some known graph theory results. Let start by recalling them, then we explain the SPE method.

3.3.1 Some graph theory information

Lemma 2. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{Z}$ a cost function.*

Then G has a cycle C of non-positive cost with respect to cost w if and only if *G has a cycle of negative cost with respect to cost w' defined by $w'(e) = \|V\| \cdot w(e) - 1$:*

$$\sum_{e\in C} w(e) \leq 0 \iff \sum_{e\in C} w'(e) < 0$$

Proof. Clearly any cycle of G of negative cost w.r.t. w is also a cycle of strictly negative cost w.r.t. w'.

Let $c = (e_1, \ldots, e_p)$ be an elementary cycle.

Then

$$\begin{aligned}
\sum_{1\leq i\leq p} w'(e_i) < 0 &\iff \|V\| \cdot \sum_{1\leq i\leq p} w(e_i) - p < 0 \\
&\iff \|V\| \cdot \sum_{1\leq i\leq p} w(e_i) < p \\
&\iff \sum_{1\leq i\leq p} w(e_i) < \frac{p}{\|V\|}
\end{aligned}$$

Hence if c is an elementary cycle of strictly negative cost w.r.t. w', i.e. if

$$\sum_{1\leq i\leq p} w'(e_i) < 0$$

Then we have

$$\sum_{1\leq i\leq p} w(e_i) < \frac{p}{\|V\|}$$

Since c is elementary, we have $p \leq \|V\|$ and thus

$$\sum_{1\leq i\leq p} w(e_i) < 1$$

And since the left hand term is an integer, we have

$$\sum_{1\leq i\leq p} w(e_i) \leq 0$$

This shows that c is a cycle of negative cost w.r.t. w. □

Lemma 3. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{R}$ a cost function.*

Then G has a cycle C of negative cost (i.e. $\sum_{e\in C} w(e) < 0$) if and only if *the constraints system $\mathcal{S}_{G,w}$ defined below is infeasible.*

$$(\mathcal{S}_{G,w}) \begin{cases} \forall e \in E :, x_{tgt(e)} - x_{src(e)} \leq w(e) \\ \forall v \in V, x_v \in \mathbb{R} \end{cases}$$

Proof. For the proof, see also [4].

- Assume that G has a cycle of negative cost.

  Let $e_1, \ldots, e_p$ be a cycle with $tgt(e_i) = src(e_{i+1})$ for $1 \leq i < p$, $tgt(e_p) = src(e_1)$ and $\sum_{1\leq i\leq p} w(e_i) < 0$

  Assume that $\mathcal{S}_{G,w}$ has a solution.

  This solution should satisfy:

$$\begin{cases} x_{tgt(e_1)} - x_{src(e_1)} \leq w(e_1) \\ x_{tgt(e_2)} - x_{src(e_2)} \leq w(e_2) \\ \vdots \\ x_{tgt(e_p)} - x_{src(e_p)} \leq w(e_p) \end{cases}$$

  Thus, by summing all these inequalities, we get

$$\sum_{1\leq i\leq p} (x_{tgt(e_i)} - x_{src(e_i)}) \leq \sum_{1\leq i\leq p} w(e_i)$$

  Thus

$$\sum_{1\leq i\leq p} (x_{tgt(e_i)} - x_{src(e_i)}) < 0$$

  But we have

$$\begin{aligned} \sum_{1\leq i\leq p} (x_{tgt(e_i)} - x_{src(e_i)}) &= \sum_{1\leq i\leq p} x_{tgt(e_i)} - \sum_{1\leq i\leq p} x_{src(e_i)} \\ &= \left(x_{tgt(e_p)} + \sum_{1\leq i<p} x_{tgt(e_i)} \right) - \left(x_{src(e_1)} + \sum_{2\leq i\leq p} x_{src(e_i)} \right) \\ &= \sum_{1\leq i<p} x_{tgt(e_i)} - \sum_{2\leq i\leq p} x_{src(e_i)} \\ &= \sum_{1\leq i<p} x_{src(e_{i+1})} - \sum_{2\leq i\leq p} x_{src(e_i)} \\ &= \sum_{2\leq i\leq p} x_{src(e_i)} - \sum_{2\leq i\leq p} x_{src(e_i)} \\ &= 0 \end{aligned}$$

  Thus $0 < 0$, which is a contradiction.

  Hence $\mathcal{S}_{G,w}$ has no solution and is thus infeasible.

- Assume that $\mathcal{S}_{G,w}$ is infeasible.

  If G has no cycle of strictly cost, then G can be modified as follows.

  Add a source $\top$ to G connected to each vertex $v \in V$ and extend w so that $w(\top, v) = 0$ for any $v \in V$.

  Then the distance from $\top$ to any other vertex v (ie the length of the shortest path from $\top$ to v) is well defined, since this extension of G has no cycle of negative cost (observe that $\top$ cannot belong to any cycle since it has no incoming edges). For instance, Bellman Ford algorithm can successfully compute this distance function.

  Let $d(\top, v)$ denote the distance from $\top$ to v for any vertex $v \in V$.

  Let $e \in E$. Then we have $d(\top, tgt(e)) \leq d(\top, src(e)) + w(e)$ by property of a distance function.

  Thus if we pose for any $v \in V$, $x_v = d(\top, v)$ then we have just found a solution of $\mathcal{S}_{G,w}$.

  This is a contradiction.

  So G has a cycle of strictly negative cost.

□

Corollary 1. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{Z}$ a cost function.*

Then G has a cycle C of non-positive cost with respect to cost w (i.e. $\sum_{e \in C} w(e) \leq 0$) if and only if the system composed of the following constraints is infeasible.

$$\forall e \in E, x_{tgt(e)} - x_{src(e)} \leq \|V\| \cdot w(e) - 1$$

where $\forall v \in V, x_v \in \mathbb{R}$.

Proof. A direct consequence of Lemma 2 and 3. □

3.3.2 Shortest path equation (SPE) method

Corollary 1 defines the heart of the SPE method. We conclude that the associated DDG is lexicographic positive *if and only if* there exists $|\mathcal{V}|$ variables $x_v \in \mathbb{R}$ for $v \in \mathcal{V}$ such that

$$\forall e \in \mathcal{E} : x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1$$

Recall that $\mathcal{V} = V \cup K$ where V is the set of vertices of the initial DDG and K is the set of all killing nodes.

We thus define a linear problem as follows.

For each vertex $v \in \mathcal{V}$, we define a *continuous* variable x_v.

For each anti-dependence edge $e = (k_u^t, v)$ corresponding to the reuse edge $e_r = (u, v)$, we define a variable $\gamma^t(u, v)$, so that the distance of e is $\lambda(e) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$.

We seek to minimise $\sum_{t \in \mathcal{T}} \alpha_t \sum_{(u,v) \in E^{\text{reuse},t}} \gamma^t(u, v)$. We also require that $\hat{\mu}^t(u, v) + \gamma^t(u, v) \geq \hat{\mu^*}^t(u, v)$ for any $(u, v) \in E^{\text{reuse},t}$, where $\hat{\mu^*}^t(u, v)$ is the solution of the scheduling problem (first step of SIRALINA). This is because $\hat{\mu^*}^t(u, v)$ defines the minimal valid values for any reuse distances.

We thus have defined a complete linear program, which contains $O(|\mathcal{V}| + |\mathcal{E}|)$ variables and $O(|\mathcal{E}|)$ equations (this is roughly $|\mathcal{E}|$ times smaller than the linear program of Figure 3.3). The linear program of the SPE method is given in Figure 3.4.

Once a solution is found for the linear program of Figure 3.4, we set the new distance of $e = (k_{u^t}, v) \in E^{\mu,t}$ as equal to $\lambda(e) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$.

As for the previous heuristic based on retiming, it is sufficient to work with a relaxed version of this problem where the type of $\gamma^t(u, v)$ is not constrained to be integer and define $\lambda(e) = \hat{\mu}^t(u, v) + \lceil \gamma^t(u, v) \rceil$. This relaxed version is shown oin Figure 3.5. Remember that for edges $e = (k_u^t, v) \in E^{\mu,t}$, corresponding to a reuse edge $e_r = (u, v) \in E^{\text{reuse},t}$, we have that $\lambda(e) = \hat{\mu}^t(u, v) + \gamma^t(u, v)$, while $\hat{\mu}^t(u, v)$ has been already computed in iterative SIRALINA (Algorithm 1): $\hat{\mu}^t(u, v) = \hat{\mu}^t_{(i-1)}(u, v)$, $E^{\text{reuse},t} = E^{\text{reuse},t}_{(i-1)}$ and the set $E^{\mu,t} = \{\Phi(e_r) \mid e_r \in E^{\text{reuse},t}_{(i-1)}\}$. Thus, we give in Figure 3.6 a more explicit version of program of Figure 3.5.

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{subject to} & \\
\forall e \in \mathcal{E}, & x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \hat{\mu}^t(u,v) + \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) \\
\forall u \in \mathcal{V}, & x_u \in \mathbb{R} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \gamma^t(u,v) \in \mathbb{Z} \\
\text{where} & \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & \lambda(e) \stackrel{\text{def}}{=} \hat{\mu}^t(u,v) + \gamma^t(u,v)
\end{cases}
$$

Figure 3.4: Linear program based on shortest paths equations (SPE)

Algorithm 3 Heuristic based on shortest paths equations

function UPDATEREUSEDISTANCES($(G, (\hat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}, (E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}})$)

Solve the linear program of Figure 3.6 to compute $(\gamma^t(u,v))$

return $(\hat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ where $\hat{\mu}^t_{(i)}(u,v) \stackrel{\text{def}}{=} \begin{cases} \hat{\mu}^t_{(i-1)}(u,v) + \lceil \gamma^t(u,v) \rceil & \text{if } (u,v) \in E^{\text{reuse},t}_{(i-1)} \\ \hat{\mu}^t_{(i-1)}(u,v) & \text{otherwise} \end{cases}$

end function

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{subject to} & \\
\forall e \in \mathcal{E}, & x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \hat{\mu}^t(u,v) + \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) \\
\forall u \in \mathcal{V}, & x_u \in \mathbb{R} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}, & \gamma^t(u,v) \in \mathbb{R} \\
\text{where} & \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & \lambda(e) \stackrel{\text{def}}{=} \hat{\mu}^t(u,v) + \gamma^t(u,v)
\end{cases}
$$

Figure 3.5: Relaxed linear program based on shortest paths equations (SPE)

$$
\begin{cases}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{subject to} & \\
\forall e \in E \cup E^k, & x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1 \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & x_{tgt(e)} - x_{src(e)} - \|\mathcal{V}\| \cdot \gamma^t(u,v) \leq \|\mathcal{V}\| \cdot \hat{\mu}^t_{(i-1)}(u,v) - 1 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(u,v) \geq \hat{\mu^*}^t(u,v) - \hat{\mu}^t_{(i-1)}(u,v) \\
\forall u \in \mathcal{V}, & x_u \in \mathbb{R} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(u,v) \in \mathbb{R}
\end{cases}
$$

Figure 3.6: Relaxed linear program based on shortest paths equations (full constraints)

Hence our implementation of $UpdateReuseDistances(G, (\hat{\mu}^t)_{t\in\mathcal{T}}, (E^{\text{reuse},t})_{t\in\mathcal{T}})$ is given by Algorithm 3.

In theory, the linear problem of Figure 3.6 is much easier to solve than the one of Figure 3.3, because it contains only continuous variables. In the next chapter, we will see that it is also the case in practice.

Chapter 4

Experimental results

In this chapter, we compare the experimental results of the reactive and proactive strategies presented in the previous chapter.

4.1 Experimental setup and environment

Regarding the proactive strategy, we have implemented the two heuristics of non-positive cycles elimination (retiming and shortest path equation) presented in Section 3. Since SPE has shown more efficient result, it have been integrated to `SIRAlib` and released as an open source software.

Our experiments have been conducted on a regular Linux workstation (Intel Xeon, 2.33 GHZ, 9 Gigabytes of memory).

The data dependency graphs used for experiments come from SPEC2000, SPEC2006, MEDIABENCH and FFMPEG sets of benchmarks. The number of benchmarks per family is the following.

MEDIABENCH	SPEC2000	SPEC2006	FFMPEG
1592	3841	1274	2030

We consider a processor architecture with NUAL semantics, involving three types of registers, namely FP (floating point registers), GR (generic registers) and BR (boolean registers). Thus $\mathcal{T} = \{\text{FP}, \text{GR}, \text{BR}\}$. The number of benchmarks per family involving these types is the following.

Type	MEDIABENCH	SPEC2000	SPEC2006	FFMPEG
FP	313	317	87	47
GR	1592	3838	1274	2030
BR	1592	3841	1274	2030

The distribution[1] of the sizes (the number of vertices) of the DDG is the following.

	MEDIAB.	SPEC2000	SPEC2006	FFMPEG
MIN	3	3	5	4
FST	10	12	16	18
MED	16	22	24	37
THD	28	28	30	111
MAX	212	163	212	783

The distribution of the number of values per register type is the following (only benchmarks that involve the considered type are taken into account).

[1] MIN stands for MINimum, FST for FirST quantile (25% of the population), MED for MEDian (50% of the population), THD for THirD quantile (75% of the population) and MAX for MAXimum

Type		MEDIAB.	SPEC2000	SPEC2006	FFMPEG
FP	MIN	1	1	1	1
	FST	2	3	3	2
	MEDIAN	4	6	4	5
	THD	8	14	12	8
	MAX	68	72	132	32
GR	MIN	1	1	1	2
	FST	6	7	8	12
	MEDIAN	9	12	12	29
	THD	16	17	18	105
	MAX	208	81	74	749
BR	MIN	1	1	1	1
	FST	1	1	1	1
	MEDIAN	1	3	3	1
	THD	3	5	4	1
	MAX	21	27	35	139

4.1.1 The frequency of non-positive cycles detection

By using the SIRALINA heuristic, we measured the proportion of DDG that became non-lexicographic positive because of periodic register constraints on NUAL semantics:

	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
Proportion	30.77%	28.16%	41.90%	92.21%

As can be seen, the above table shows that the problem of non-positive cycles is not rare in practice. While a reactive approach may be beneficial (tolerate the problem then correct it), the frequency of the problem is a justification for our proactive strategies.

4.1.2 Description of the experiments

Heuristics nomenclature

Our methods to avoid the creation of non-positive cycles are of three sorts:

1. UAL is the (pessimistic) naive heuristic which consists in solving SIRALINA with an UAL semantics only. That is, we do not consider NUAL code semantics from the beginning.
2. CHECK is the reactive strategy which consists in firstly applying SIRALINA with NUAL semantics. If a non-positive cycle is detected, we apply a second pass, which apply SIRALINA but with a UAL semantics.
3. For the proactive strategy, we implemented our two heuristics:
   (a) RET is the heuristic based on circuit retiming ideas. If $n-1 \geq 0$ is the bound on the maximal number of iterations used, we write RET_n. Hence RET_n means that the **repeat-until** loop of Algorithm 1 has been executed at most n times (and at least one).
   (b) SPE is the heuristic based on shortest paths equations. If $n-1 \geq 0$ is the bound on the maximal number of iterations used, we write SPE_n.

Empirical efficiency measures

For each heuristic of non-positive cycle elimination, for each DDG, for each initiation interval II between MII and L^2, we measured the execution time taken by Iterative SIRALINA (Algorithm 1) to reduce register pressure of the given DDG; we recorded also the number of registers computed by Iterative SIRALINA. We are going to examine these results in the next sections.

To put these results in a real world context, we have also modelled three possible target architectures by setting the following register constraints (number of available registers).

[2] L is an upper bound on the admissible values for II

Name of the architecture	FP registers	GR registers	BR registers
Small architecture	32	32	4
Medium architecture	64	64	8
Large architecture	128	128	8

When the number of available registers is fixed in the architecture, we may need to iterate on multiple values for II in order to get a solution below the processor capacity; that is, since register minimisation is applied for a fixed II, we may need to iterate on multiple values of II if the minimised register requirement is still above the number of available registers. The strategy for iterating over II is the following:

- Check whether SIRALINA[3] produces a solution that satisfies the register constraints for II = MII.
  - if yes, stop and return the solution.
  - if no, check whether SIRALINA gives a solution that satisfies the constraints for II = L (maximal allowed value for II).
    - if yes, search linearly the smallest II > MII such that SIRALINA computes a solution that satisfies the register constraints.
    - if no, then fail (no solution found, spilling is necessary).

For each architecture, for each heuristic, and for each DDG G, we determined whether SIRALINA is able to find a solution that satisfies the architecture constraints. We thus measured:

- whether a solution exists or not;
- the elapsed time needed to determine whether a solution exists;
- the smallest II for which a solution exists (when applicable).

Regarding the iterative heuristic of non-positive cycles elimination (Algorithm 1), we arbitrary fixed the maximal number of iterations to 3 and 5. As we shall see in the following, the heuristic based on circuit retiming tends to converge[4] quickly in terms of number of iterations, so we also did the experiments with this heuristic by setting the maximal number of iterations to 1. In order to get an idea of how many iterations the iterative methods could take in the worst case, we also did the experiments by settings the maximal allowed number of iterations to 1000 and recorded the reached number of iterations. Remember that if a fixed point (convergence) is detected, the iterative algorithm stops before reaching 1000 iterations.

4.2 Comparison of the heuristics execution times

In this section, we compare and comment the execution times of the heuristics of non-positive cycles elimination.

4.2.1 Time to minimise register pressure for a fixed II

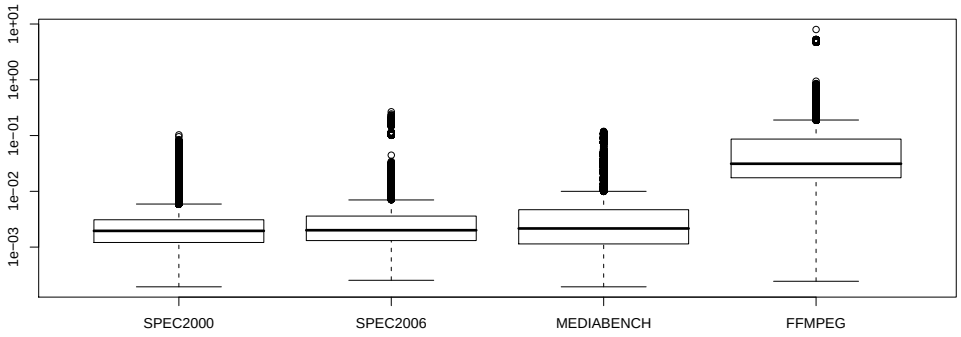
In this section, we solve Iterative SIRALINA with all values of II. Figure 4.1 shows the distribution of execution times of UAL heuristic. We use boxplot to graphically depicts the quartiles of the distribution.

Figure 4.2 shows the distribution of execution times of CHECK heuristic.

Regarding RET_n heuristic, we only have partial results because unfortunately, this heuristic is intractable in practice. We summarise in the table below the number of DDG (per benchmark family) for which we managed to get experimental measures with RET heuristics. Figure 4.3 shows the distribution of execution times of RET_n heuristic for $n \in \{1, 3, 5, 1000\}$. Since we have partial results only, we observe that the maximal execution time with 1 iteration is higher than the one obtained with 3 iterations.

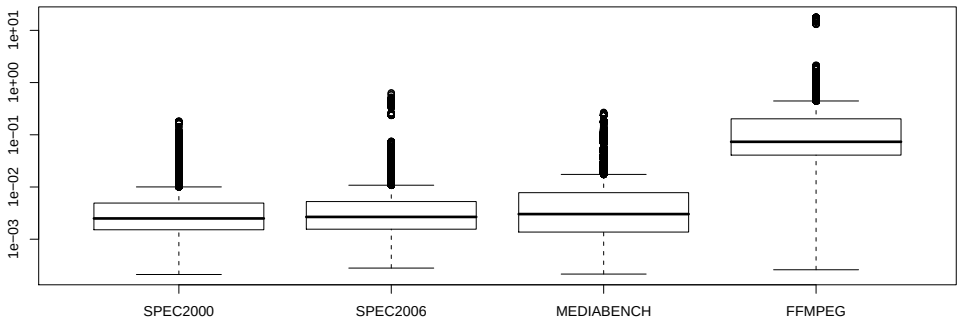
[3]Here, any of the variants of SIRALINA can be used: UAL, CHECK, RET or SPE. If the variant is RET or SPE, then the used algorithm becomes Iterative SIRALINA instread of SIRALINA.

[4]The term of convergence here means that the reuse distances do not change in two successive iterations of our iterative methods.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000194	0.000254	0.000194	0.000244
FST	0.00121	0.001309	0.00114	0.017507
MEDIAN	0.001954	0.002007	0.002158	0.031229
THD	0.003092	0.003601	0.004682	0.08647
MAX	0.102878	0.267225	0.118746	7.97499

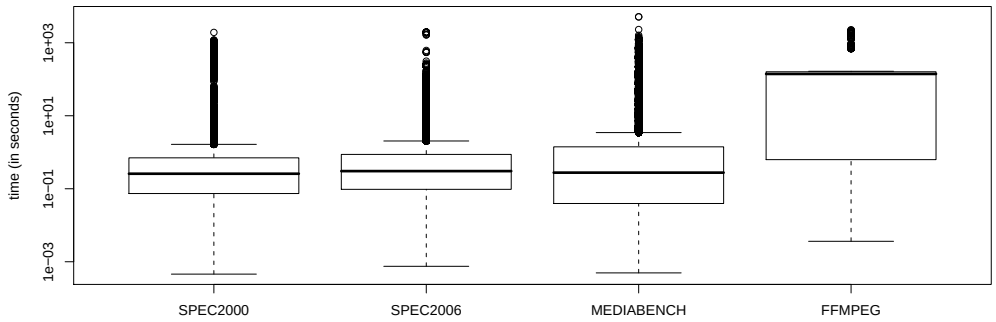
Figure 4.1: Execution times of UAL (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000211	0.00028	0.000215	0.00026
FST	0.001517	0.001556	0.00137	0.040792
MEDIAN	0.002499	0.002673	0.003029	0.073375
THD	0.004925	0.005263	0.007788	0.202154
MAX	0.183653	0.636804	0.268994	17.8744

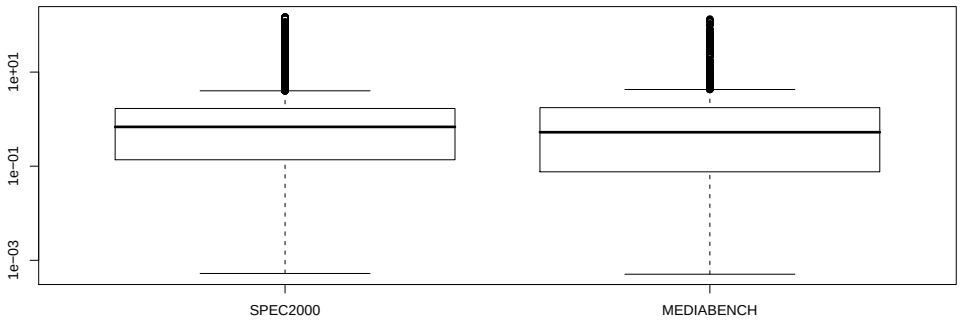
Figure 4.2: Execution times of CHECK (in seconds)

	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
RET_1	1289	1206	1373	22
RET_3	1251	0	322	0
RET_5	1251	0	322	0
RET_{1000}	1251	0	322	0



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000458	0.00075	0.000495	0.003633
FST	0.073903	0.096325	0.039457	0.62724
MEDIAN	0.257963	0.304235	0.276409	139.927
THD	0.703017	0.87254	1.40718	160.321
MAX	1920.95	1980.08	5142.86	2272.67

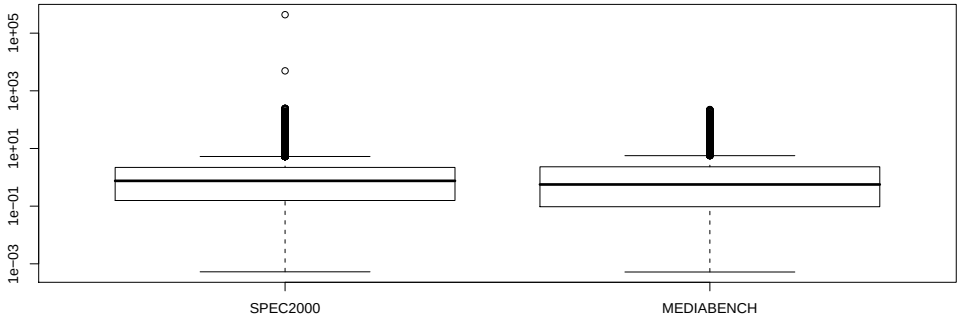
(a) 1 iteration



	SPEC2000	MEDIABENCH
MIN	0.000525	0.000506
FST	0.137005	0.075961
MEDIAN	0.684768	0.526998
THD	1.68522	1.75397
MAX	148.937	134.729

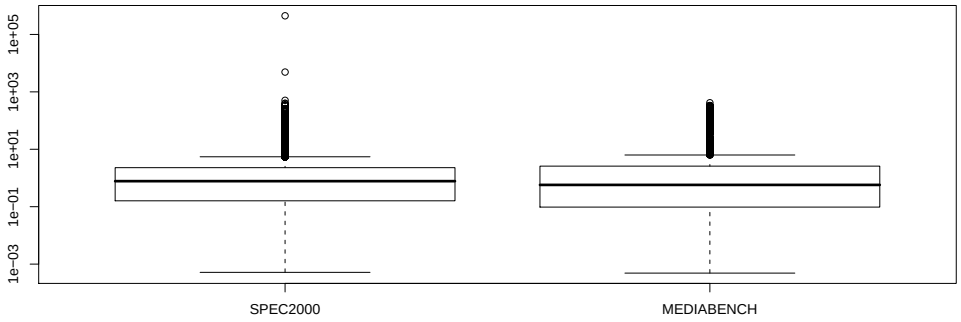
(b) 3 iterations

Figure 4.3: Execution times of RET (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000528	0.00052
FST	0.156939	0.095541
MEDIAN	0.753487	0.56428
THD	2.20552	2.31773
MAX	437291	224.419

(c) 5 iterations

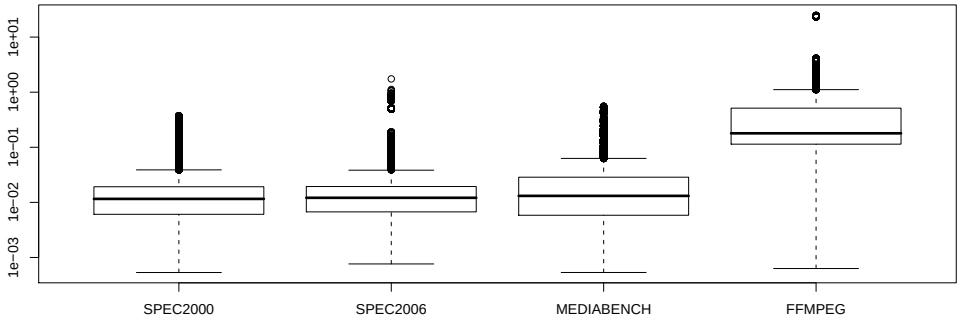


	SPEC2000	MEDIABENCH
MIN	0.000516	0.000486
FST	0.160326	0.096791
MEDIAN	0.775337	0.57392
THD	2.28936	2.58837
MAX	446469	417.676

(d) 1000 iterations

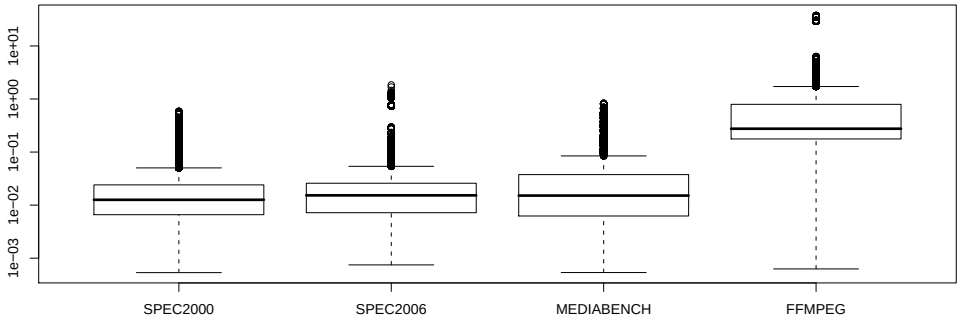
Figure 4.3: Execution times of RET (in seconds)

Figure 4.4 shows the distribution of execution times of SPE_n heuristic for $n \in \{3, 5, 1000\}$. Contrary to RET heuristic, we have a full set of results here.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000533	0.000763	0.000534	0.00063
FST	0.006075	0.006724	0.00584	0.113908
MEDIAN	0.011568	0.012111	0.013119	0.17934
THD	0.019208	0.019398	0.028636	0.51334
MAX	0.375687	1.74257	0.556245	24.9619

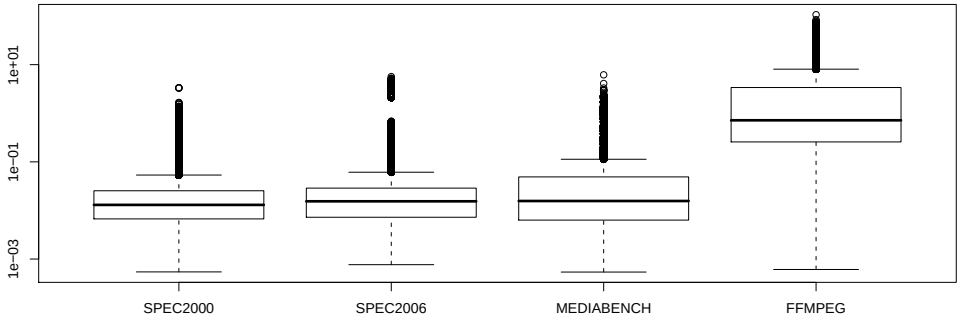
(a) 3 iterations



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000534	0.000747	0.000536	0.000627
FST	0.006597	0.007188	0.006236	0.176632
MEDIAN	0.012591	0.015273	0.015129	0.275262
THD	0.024067	0.025851	0.037576	0.793465
MAX	0.587562	1.84686	0.834527	37.7923

(b) 5 iterations

Figure 4.4: Execution times of SPE (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000544	0.000766	0.00054	0.00061
FST	0.006702	0.007244	0.00633	0.257014
MEDIAN	0.013011	0.015423	0.015643	0.716324
THD	0.02542	0.028811	0.048997	3.3835
MAX	3.35579	5.67563	6.17455	106.426

(c) 1000 iterations

Figure 4.4: Execution times of SPE (in seconds)

From the above results, we see as expected that UAL is the fastest heuristic. CHECK is between one and three times slower than UAL, which was also expected because it consists in running SIRALINA, performing a check and in the worst case running SIRALINA a second time.

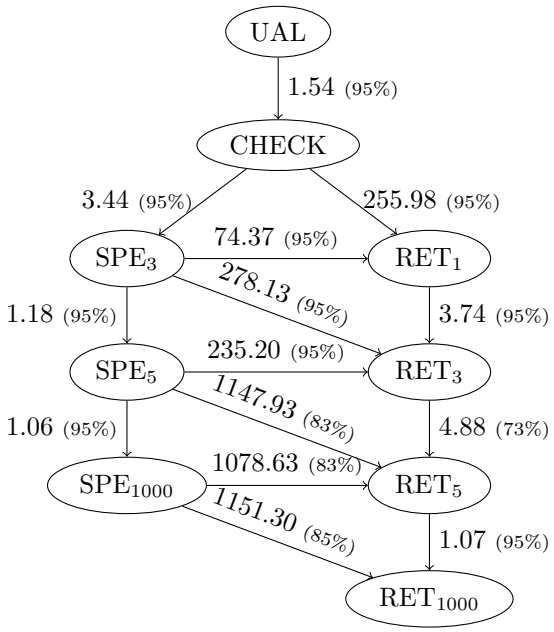
Regarding our proactive heuristics of non-positive cycles elimination, we observe that RET is not really usable in practice because the execution times are quite prohibitive, even for one single iteration (median execution time reaches already 0.25 seconds). For this reason, we were unable to run the whole set of benchmarks with this heuristic.

In the following, all the results involving RET heuristics are only available on a partial sets of the benchmarks, whose size is given in the table above.

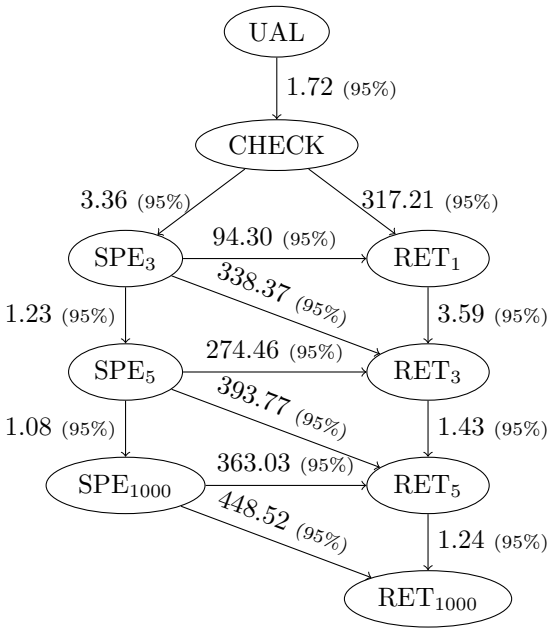
On the contrary, SPE heuristic seems to have a quite reasonable running time, but is yet sensibly more expensive than UAL or CHECK (about 10 times slower).

Statistical analysis of the execution times

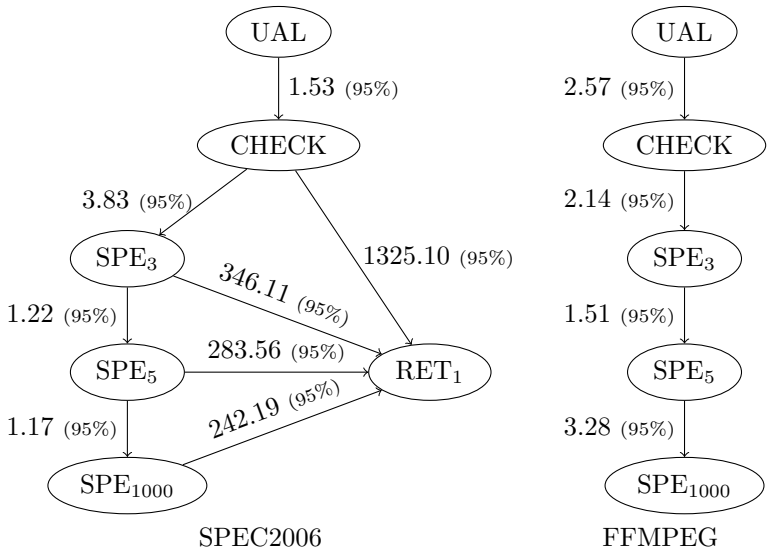
We compared each paired list of measures with the Student t-test. Results are represented below by graphs. A directed edge between two nodes means that the heuristic of the source node is statistically faster than the heuristic of the target node. The edges are labelled with the level of confidence (between parenthesis) of the statistical t-test and the mean speedup ratio. For instance, on SPEC2000 benchmarks, UAL is about 1.54 times faster than CHECK with a confidence level of 95%.



SPEC2000



MEDIABENCH



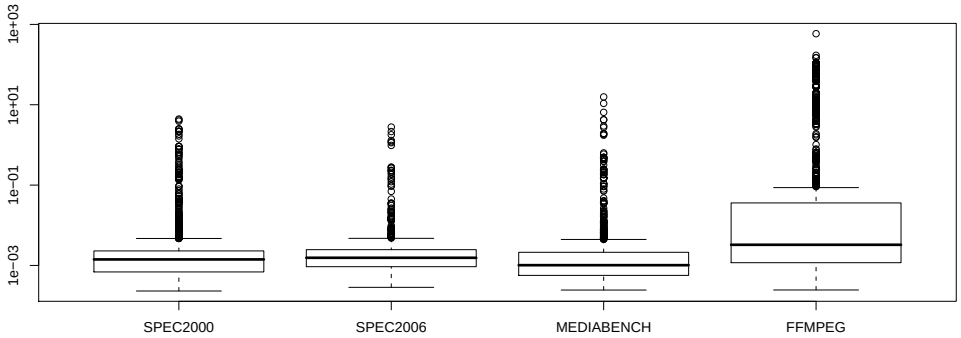
These analysis confirm what we observed previously:

- UAL is the fastest heuristic
- CHECK is the second fastest heuristic
- SPE family of heuristics are tractable: depending on the maximal number of iterations, they are between 5 and 30 times slower than UAL heuristic.
- RET family of heuristics are not usable in practice: RET_1 is already more than 800 times slower than UAL heuristic! And we were unable, in a reasonable amount of time, to reduce register pressure of the DDG coming from the FFMPEG set of benchmark.

4.2.2 Running times needed to reduce the register pressure below the architectural capacity

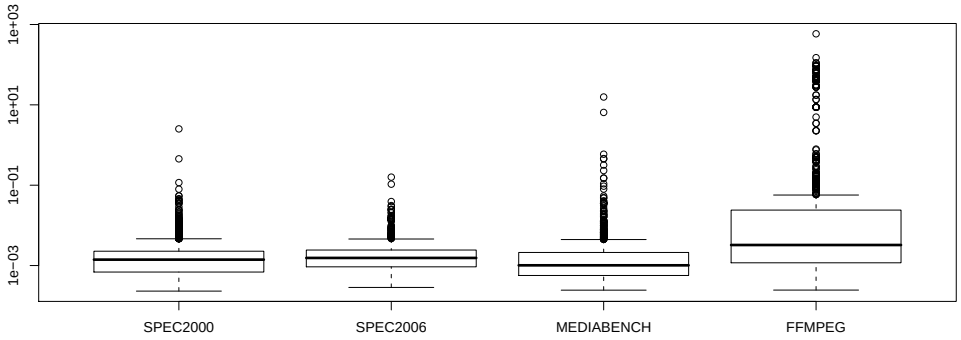
For each set of architecture constraints and for each DDG, we measured the time needed by each heuristic (UAL, CHECK, Iterative SIRALINA with RETor SPE) to satisfy the register constraints for each the three architectures (small, medium and large). Given the number of available registers, SIRALINA may iterate over II values to reach a solution below the processor capacity.

Figure 4.5 shows the distribution of execution times of UAL heuristic.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000231	0.000285	0.000244	0.000245
FST	0.000689	0.000931	0.000566	0.00117
MEDIAN	0.001412	0.001549	0.001014	0.003264
THD	0.002299	0.002469	0.002123	0.035973
MAX	4.414726	2.782836	15.68271	587.3721

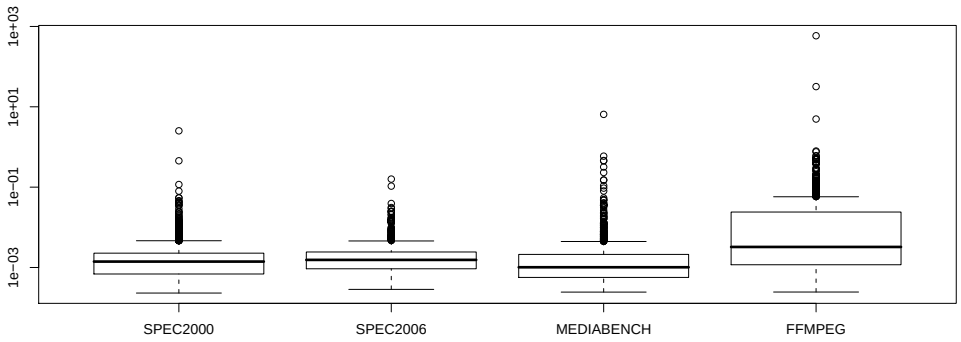
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000231	0.000285	0.000244	0.000245
FST	0.000689	0.000929	0.000566	0.00117
MEDIAN	0.001403	0.001546	0.001012	0.003245
THD	0.002274	0.002432	0.002114	0.024008
MAX	2.522404	0.158739	15.68271	587.3721

(b) medium architecture

Figure 4.5: Execution times of UAL (in seconds)

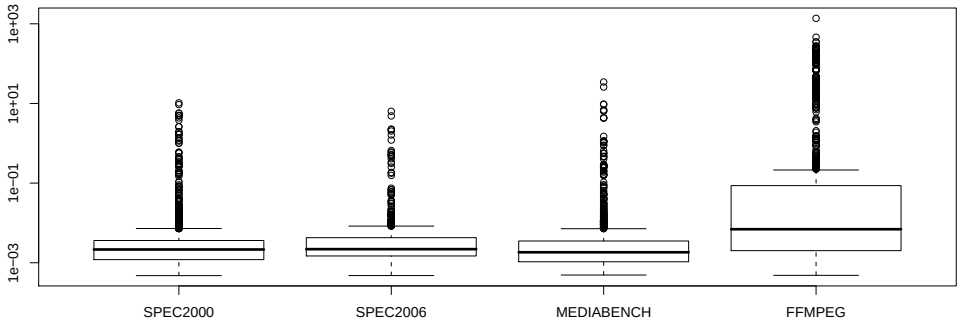


	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000231	0.000285	0.000244	0.000245
FST	0.000689	0.000929	0.000566	0.00117
MEDIAN	0.001403	0.001546	0.001012	0.003245
THD	0.002274	0.002432	0.002114	0.024008
MAX	2.522404	0.158739	6.482204	587.3721

(c) large architecture

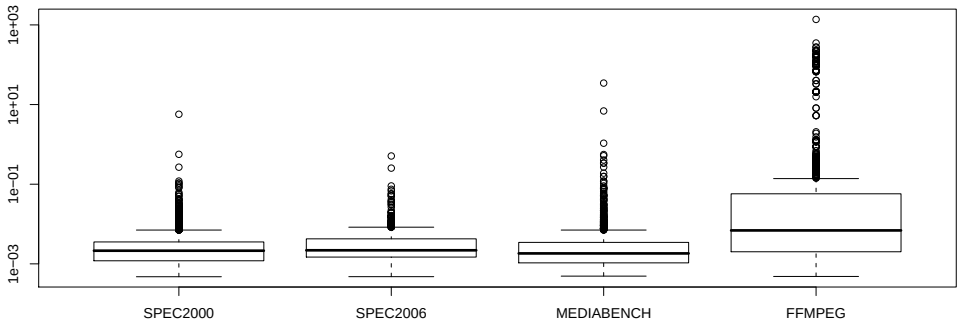
Figure 4.5: Execution times of UAL (in seconds)

Figure 4.6 shows the distribution of execution times of CHECK heuristic.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000475	0.000477	0.000492	0.000483
FST	0.001198	0.001481	0.001059	0.002012
MEDIAN	0.002152	0.002193	0.001833	0.006946
THD	0.003622	0.004264	0.003517	0.08652
MAX	10.36903	6.363512	34.6431	1373.660

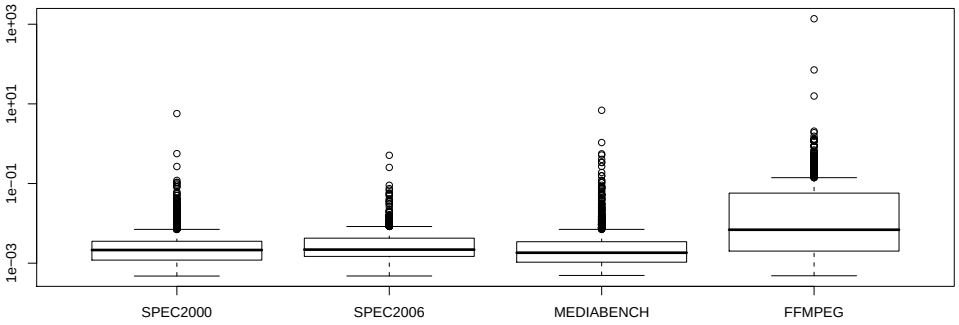
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000475	0.000477	0.000492	0.000483
FST	0.001195	0.00148	0.001059	0.002012
MEDIAN	0.002139	0.002192	0.001827	0.00691
THD	0.003552	0.004249	0.003454	0.057432
MAX	5.711776	0.511348	34.6431	1373.660

(b) medium architecture

Figure 4.6: Execution times of CHECK (in seconds)

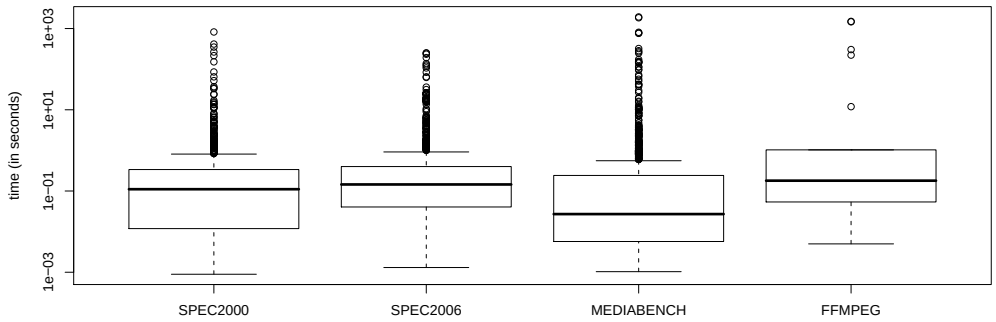


	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000475	0.000477	0.000492	0.000483
FST	0.001195	0.00148	0.001059	0.002012
MEDIAN	0.002139	0.002192	0.001827	0.00691
THD	0.003552	0.004249	0.003454	0.057432
MAX	5.711776	0.511348	6.904356	1373.660

(c) large architecture

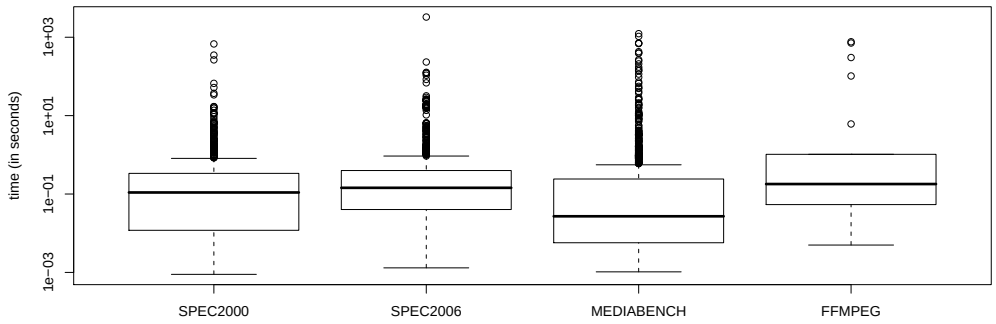
Figure 4.6: Execution times of CHECK (in seconds)

Figure 4.7 (resp. Figure 4.8, Figure 4.9, Figure 4.10) shows the distribution of execution times of RET_1 (resp. RET_3, RET_5, RET_{1000}). As previously, we only have a partial set of results for this family of heuristics.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000892	0.001306	0.001033	0.00499
FST	0.011864	0.040435	0.005711	0.053743
MEDIAN	0.110711	0.144264	0.027025	0.118959
THD	0.337554	0.399991	0.241679	1.03013
MAX	824.5258	252.596	1932.328	1500.781

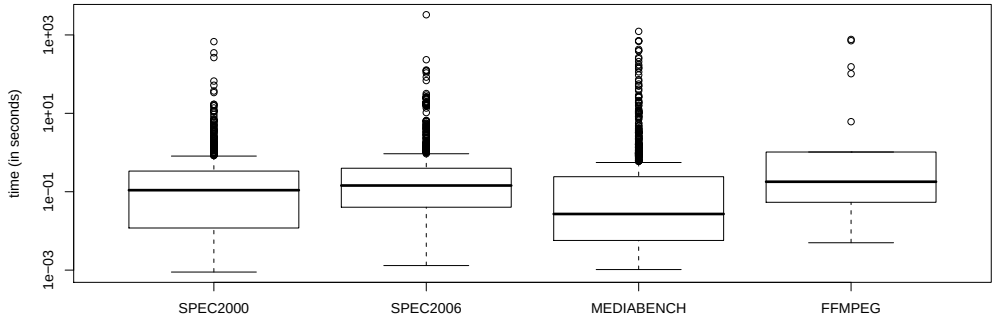
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000892	0.001306	0.001033	0.00499
FST	0.011864	0.040233	0.005711	0.053743
MEDIAN	0.1095	0.143635	0.027025	0.118959
THD	0.336481	0.397334	0.241679	1.03013
MAX	675.035	3278.032	1233.17	766.878

(b) medium architecture

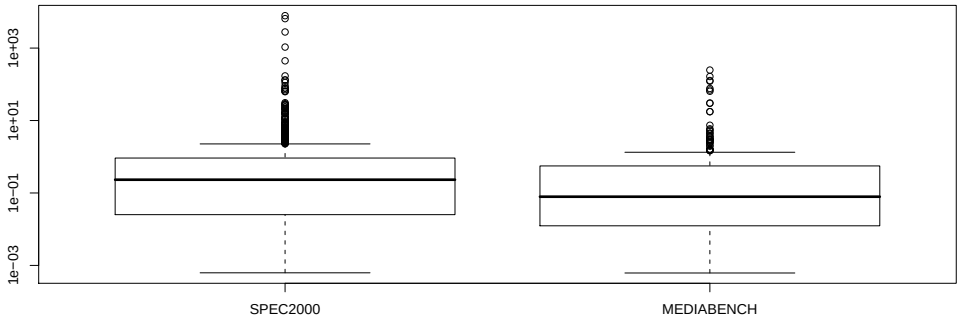
Figure 4.7: Execution times of RET_1 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000892	0.001306	0.001033	0.00499
FST	0.011864	0.040233	0.005711	0.053743
MEDIAN	0.1095	0.143635	0.027025	0.118959
THD	0.336481	0.397334	0.241679	1.03013
MAX	675.035	3278.032	1233.17	766.878

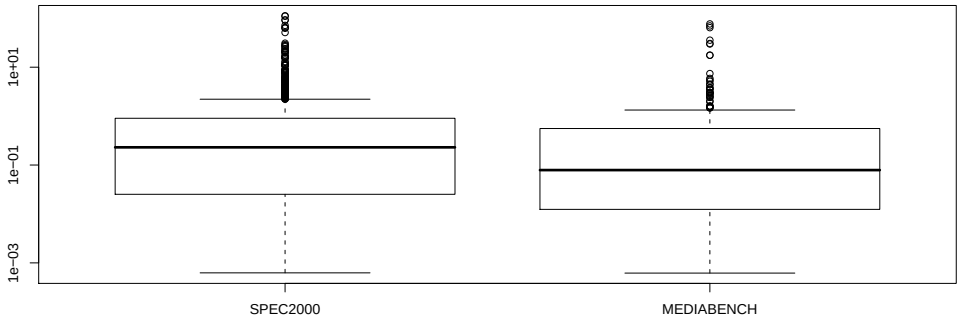
(c) large architecture

Figure 4.7: Execution times of RET_1 (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000625	0.000618
FST	0.025027	0.012009
MEDIAN	0.232303	0.077077
THD	0.921006	0.556633
MAX	7850.001	247.3721

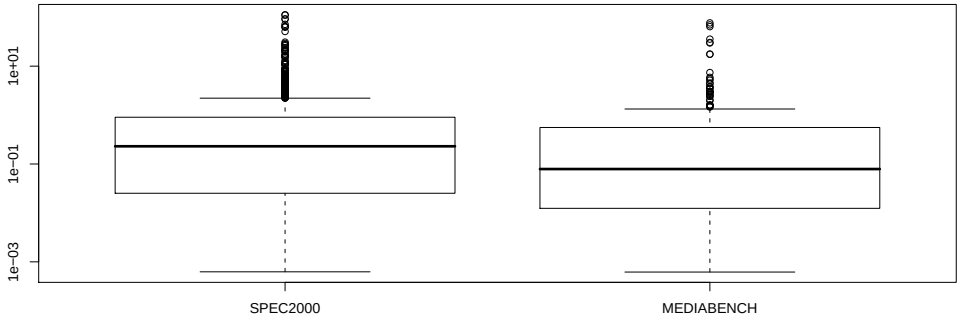
(a) small architecture



	SPEC2000	MEDIABENCH
MIN	0.000625	0.000618
FST	0.025027	0.012009
MEDIAN	0.230102	0.077077
THD	0.908128	0.556633
MAX	112.562	76.7583

(b) medium architecture

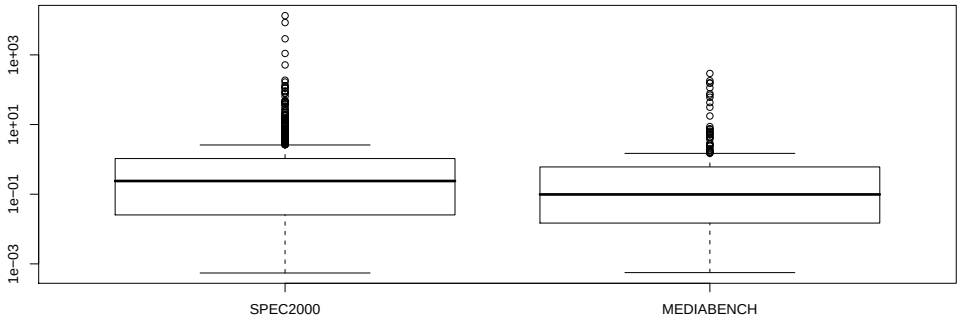
Figure 4.8: Execution times of RET_3 (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000625	0.000618
FST	0.025027	0.012009
MEDIAN	0.230102	0.077077
THD	0.908128	0.556633
MAX	112.562	76.7583

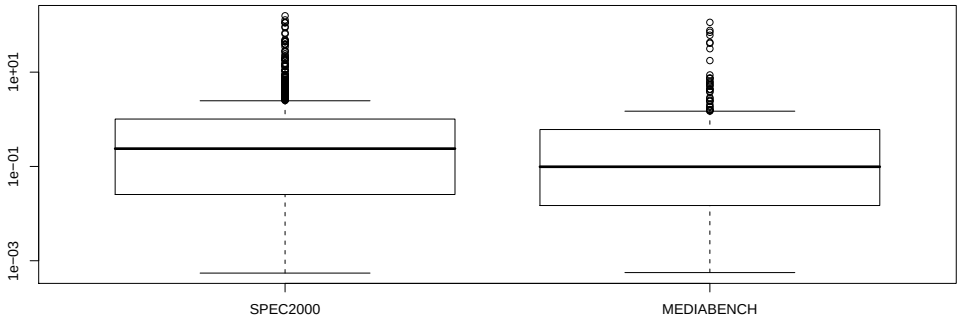
(c) large architecture

Figure 4.8: Execution times of RET_3 (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000547	0.000561
FST	0.025372	0.014658
MEDIAN	0.239417	0.09718
THD	1.05853	0.606974
MAX	13361.19	293.3859

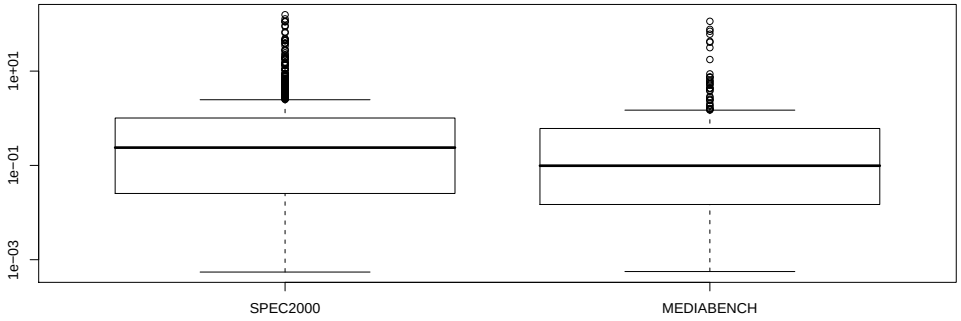
(a) small architecture



	SPEC2000	MEDIABENCH
MIN	0.000547	0.000561
FST	0.025372	0.014658
MEDIAN	0.238454	0.09718
THD	1.01617	0.606974
MAX	156.918	113.94

(b) medium architecture

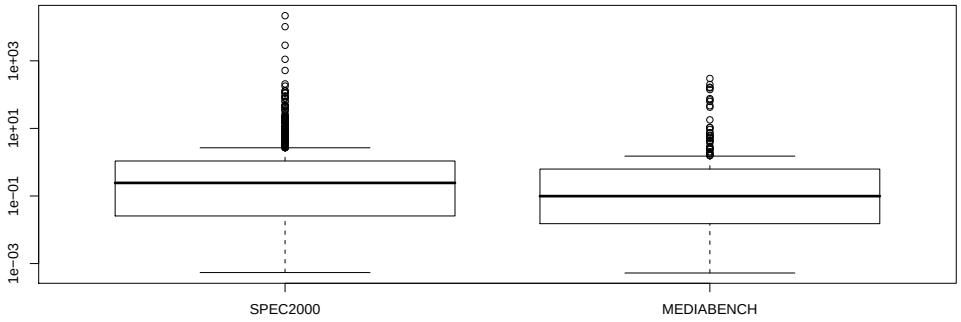
Figure 4.9: Execution times of RET_5 (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000547	0.000561
FST	0.025372	0.014658
MEDIAN	0.238454	0.09718
THD	1.01617	0.606974
MAX	156.918	113.94

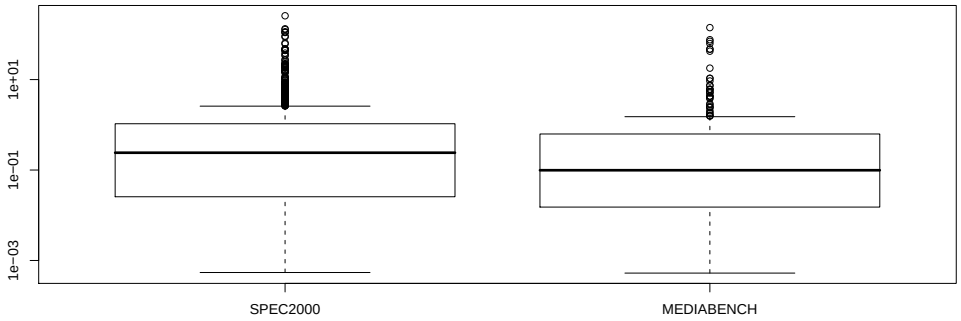
(c) large architecture

Figure 4.9: Execution times of RET_5 (in seconds)



	SPEC2000	MEDIABENCH
MIN	0.000542	0.000526
FST	0.025607	0.015036
MEDIAN	0.243903	0.098753
THD	1.08722	0.60848
MAX	21702.76	300.4762

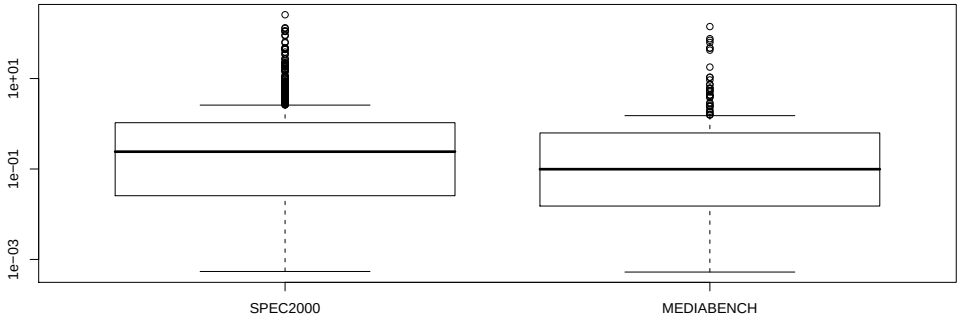
(a) small architecture



	SPEC2000	MEDIABENCH
MIN	0.000542	0.000526
FST	0.025607	0.015036
MEDIAN	0.241448	0.098753
THD	1.05805	0.60848
MAX	257.756	141.115

(b) medium architecture

Figure 4.10: Execution times of RET_{1000} (in seconds)

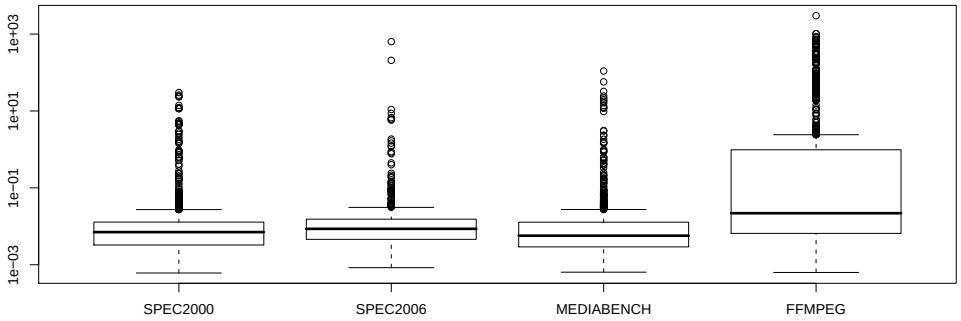


	SPEC2000	MEDIABENCH
MIN	0.000542	0.000526
FST	0.025607	0.015036
MEDIAN	0.241448	0.098753
THD	1.05805	0.60848
MAX	257.756	141.115

(c) large architecture

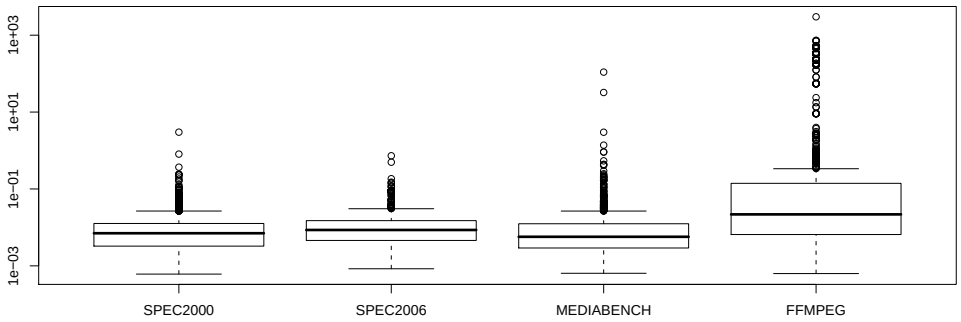
Figure 4.10: Execution times of RET_{1000} (in seconds)

Figure 4.11 (resp. Figure 4.12, Figure 4.13) shows the distribution of execution times of SPE_3 (resp. SPE_5, SPE_{1000}).



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.00061	0.00084	0.000641	0.00063
FST	0.003262	0.004581	0.002914	0.006534
MEDIAN	0.007063	0.008585	0.005698	0.021887
THD	0.01287	0.015328	0.012785	0.978112
MAX	30.14617	635.2672	109.5563	3009.366

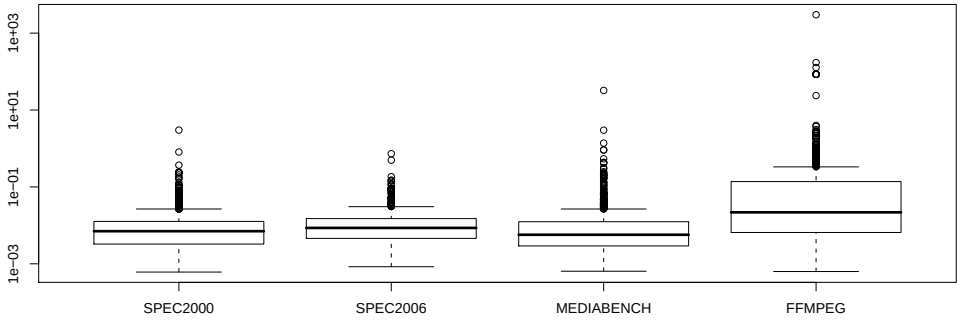
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.00061	0.00084	0.000641	0.00063
FST	0.003259	0.004581	0.002914	0.006525
MEDIAN	0.00705	0.008576	0.005697	0.021816
THD	0.012668	0.015001	0.012424	0.140078
MAX	3.003138	0.730707	109.5563	3009.366

(b) medium architecture

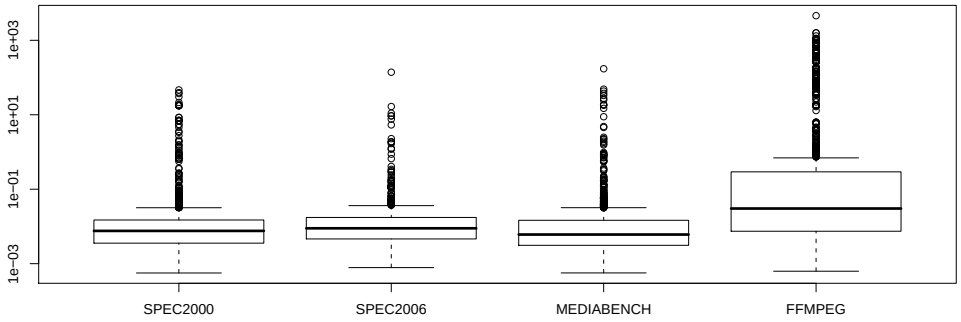
Figure 4.11: Execution times of SPE_3 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.00061	0.00084	0.000641	0.00063
FST	0.003259	0.004581	0.002914	0.006525
MEDIAN	0.00705	0.008576	0.005697	0.021816
THD	0.012668	0.015001	0.012424	0.137695
MAX	3.003138	0.730707	32.33972	3009.366

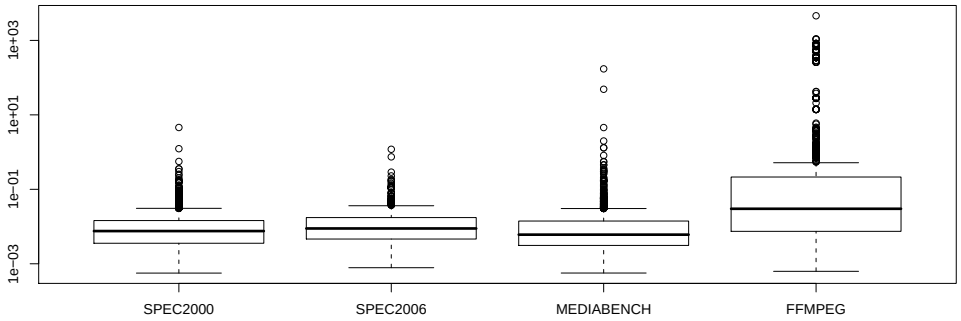
(c) large architecture

Figure 4.11: Execution times of SPE_3 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000562	0.000784	0.000564	0.000627
FST	0.003559	0.004621	0.003111	0.007435
MEDIAN	0.007589	0.008895	0.006061	0.030277
THD	0.014907	0.017438	0.014614	0.292832
MAX	46.49353	139.0469	172.0614	4591.242

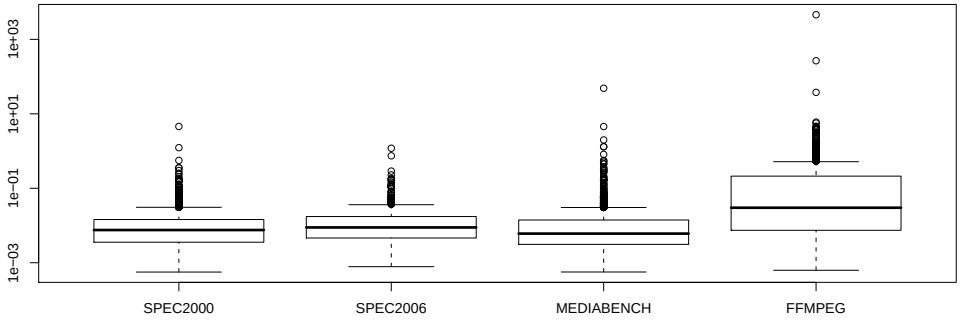
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000562	0.000784	0.000564	0.000627
FST	0.003559	0.004621	0.003111	0.007435
MEDIAN	0.007564	0.008865	0.006057	0.029973
THD	0.014489	0.017418	0.014051	0.214133
MAX	4.582993	1.19043	172.0614	4591.242

(b) medium architecture

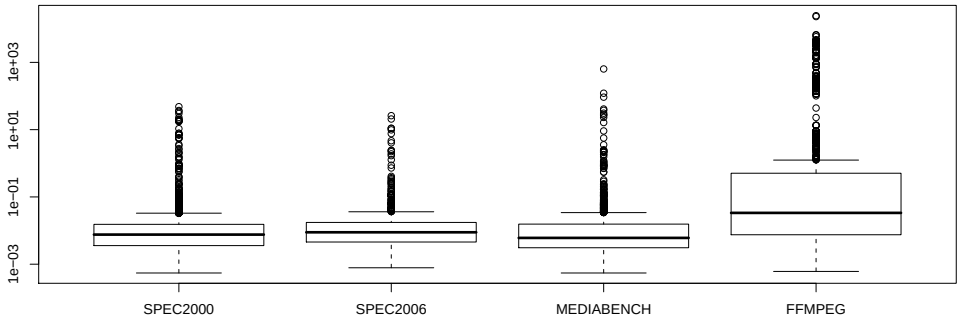
Figure 4.12: Execution times of SPE_5 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000562	0.000784	0.000564	0.000627
FST	0.003559	0.004621	0.003111	0.007435
MEDIAN	0.007564	0.008865	0.006057	0.029973
THD	0.014489	0.017418	0.014051	0.212237
MAX	4.582993	1.19043	48.74887	4591.242

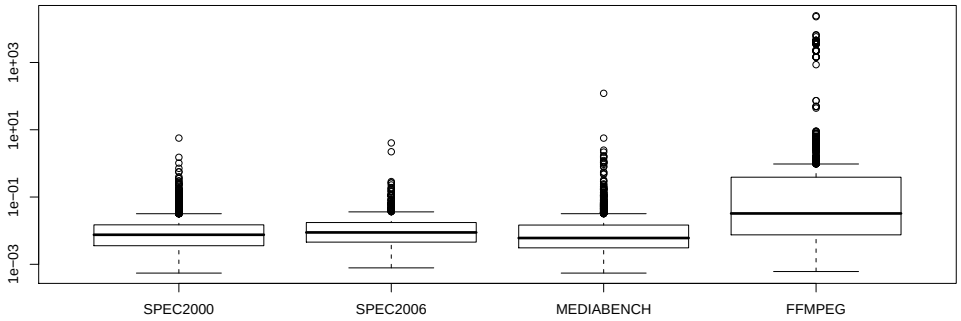
(c) large architecture

Figure 4.12: Execution times of SPE_5 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000548	0.000783	0.000548	0.00061
FST	0.003572	0.004568	0.003077	0.007507
MEDIAN	0.007593	0.008875	0.006049	0.033534
THD	0.015311	0.017456	0.015554	0.506624
MAX	48.19249	26.06242	637.3817	24472.01

(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000548	0.000783	0.000548	0.00061
FST	0.003571	0.004568	0.003077	0.007507
MEDIAN	0.007556	0.008855	0.006047	0.032467
THD	0.014908	0.017448	0.014676	0.388798
MAX	5.660698	4.06088	121.1336	24472.01

(b) medium architecture

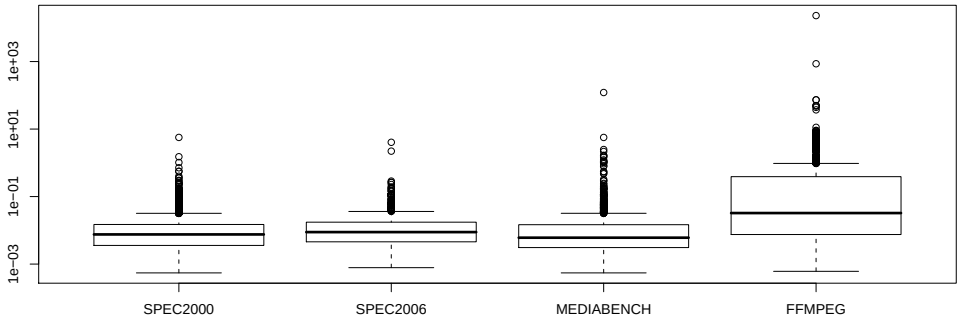
Figure 4.13: Execution times of SPE_{1000} (in seconds)

Comments

We observe, as expected, that the stronger the architecture constraints are, the slower the heuristics perform.

To ease the comparison between the different heuristics, we have computed in Figure 4.14 the ratio between the mean time of each heuristic and the mean time of UAL heuristic.

For instance, we read from these tables that CHECK is in average 1.62 times slower than UAL on SPEC2000 benchmarks when considering the small architecture.



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000548	0.000783	0.000548	0.00061
FST	0.003571	0.004568	0.003077	0.007507
MEDIAN	0.007556	0.008855	0.006047	0.032467
THD	0.014908	0.017448	0.014676	0.388798
MAX	5.660698	4.06088	121.1336	23039.27

(c) large architecture

Figure 4.13: Execution times of SPE_{1000} (in seconds)

	UAL	CHECK	SPE_3	SPE_5	SPE_{1000}	RET_1	RET_3	RET_5	RET_{1000}
Small architecture									
SPEC2000	1	1.62	5.03	5.96	7.25	994.51	847.90	1110.65	1249.29
SPEC2006	1	1.66	57.92	41.54	7.73	757.14	NA	NA	NA
MEDIABENCH	1	1.97	10.86	15.42	10.98	2094.31	549.43	743.90	960.31
FFMPEG	1	2.46	14.70	16.39	43.02	4207.15	NA	NA	NA
Medium architecture									
SPEC2000	1	1.68	5.38	6.36	7.17	775.41	800.12	1053.10	1170.92
SPEC2006	1	1.68	5.52	6.61	8.30	1715.62	NA	NA	NA
MEDIABENCH	1	1.95	5.71	7.34	10.68	2186.00	702.39	942.32	1213.24
FFMPEG	1	2.41	7.24	7.68	8.93	776.23	NA	NA	NA
Large architecture									
SPEC2000	1	1.68	5.38	6.36	7.17	775.41	800.12	1053.10	1170.92
SPEC2006	1	1.68	5.52	6.61	8.30	1715.62	NA	NA	NA
MEDIABENCH	1	1.95	5.70	7.30	10.79	1928.23	709.21	951.46	1225.02
FFMPEG	1	2.49	19.47	8.13	29.09	2829.17	NA	NA	NA

Figure 4.14: Ratio of the mean time of each heuristic over the mean time of UAL

First, we see that the heuristics perform better on SPEC2000 and SPEC2006 benchmarks rather than on MEDIABENCH or (worst) on FFMPEG benchmarks. This is not surprising because, as stated in Section 4.1, the amount of DDG to be corrected (having non-positive cycles) is low on SPEC2000 and SPEC2006 benchmarks (about 30%), higher on MEDIABENCH benchmarks (about 42%) and very high on FFMPEG benchmarks (more than 92%).

As we previously noted, CHECK heuristic is between one and three times slower than UAL heuristic.

These ratios also confirm that RET heuristics are very expensive whereas SPE heuristics are not.

At first sight, we may be astonished by the ratios produced by the iterative methods. Indeed, we may have expected that the more the number of iterations are performed, the slower the heuristic is. However, this simple reasoning does not take into account the fact that by performing more iterations for a fixed II, the heuristic may produce a *better* result, *i.e.* produce a DDG that satisfies the register constraints of the architecture. Hence, even if a single run is slower for a fixed II, it may help to find a lower II such that the register constraints are satisfied. For instance, on the SPEC2006 benchmarks under the small architecture constraints, we see that SPE_3 is about 57.92 times slower than UAL whereas SPE_5 is 41.54 times slower than UAL and SPE_{1000} is only 7.73 times slower than UAL. In other words, the more we iterate, the cheapest we are! Of course, it is not always the case. Thus, on FFMPEG benchmarks under small architecture constraints, it is worth the price to iterate (up to) 5 times rather than 3 times SPE heuristic, but it is not to iterate (up to) 1000 times rather than 5 times.

4.2.3 Convergence of the iterative proactive heuristics

We study in this section the speed of convergence (in terms of number of iterations) of RET and SPE heuristics. Recall that the iterative algorithm is said to *converge* when it reaches a fixed point, *i.e.* when the set of reuse edges does not change between two consecutive iterations of Algorithm 1. All the values of II are tested, so the experiments we consider in this section are for all DDG, for all II values, for all SIRALINA variants.

Observed results

Figure 4.15 shows the distribution of the number of iterations of RET heuristic (truncated at 1000). Again, recall that these results are partial because the heuristic did not succeed in solving all DDG instances.

Figure 4.16 shows the distribution of the number of iterations of SPE heuristic (truncated at 1000). This heuristic succeeds to solve all DDG instances.

Comments on the observed number of iterations

We see from these results that RET heuristic seems to converge faster than SPE heuristic.

We also observe that on a few number of DDG, the upper bound of 1000 iterations has been reached by SPE heuristic. It is indeed well possible that the iterative process does not terminate in the general case (we have not observed this for RET heuristic because we only have partial results but conjecture it might also happen).

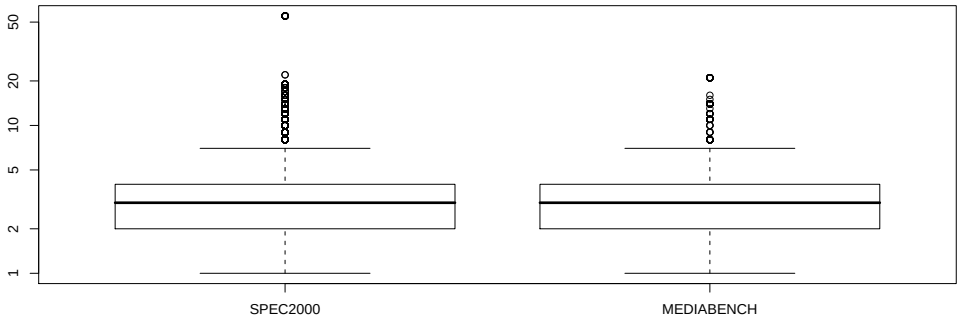
Note finally that this information may be used to set in an industrial compiler the upper bound on the maximal number of iterations.

4.3 Qualitative analysis of the heuristics

In this section, we study the quality of the solution produced by the heuristics. The qualitative aspects include the number of registers needed to schedule the DDG and the loss of parallelism due to an increase of the MII resulted from SIRALINA (with its variants UAL, CHECK, RETand SPE).

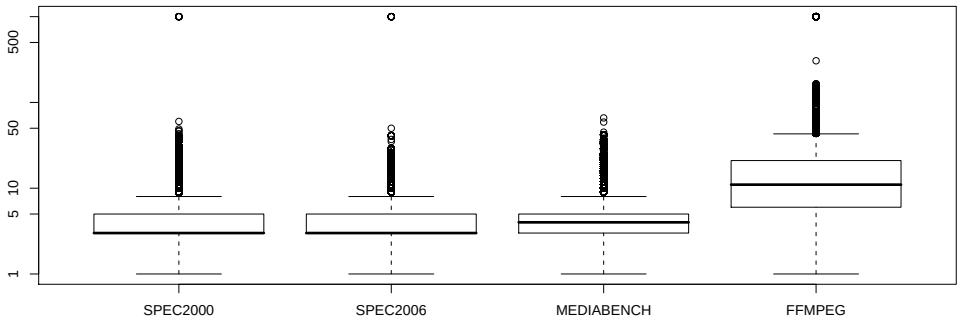
4.3.1 Number of saved registers

In this section, we analyse the number of registers each heuristic manage to optimise. Our tests are for all DDG, for all II values, for all SIRALINA variants.



	SPEC2000	MEDIABENCH
MIN	1	1
FST	2	2
MEDIAN	3	3
THD	4	4
MAX	55	21

Figure 4.15: Maximum number of iterations for RET



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	1	1	1	1
FST	3	3	3	6
MEDIAN	3	3	4	11
THD	5	5	5	21
MAX	1000	1000	66	1000

Figure 4.16: Maximum number of iterations for SPE

We compare graphically the heuristics: for each set of benchmarks, and each register types, we construct a partial order (lattice) as follows:

- the vertices are labelled with the name of the heuristic
- a directed edge links an heuristic A to an heuristic B iff the number of registers (of considered type) computed by heuristic B is statistically greater (worse) than the number of registers (of the same type) computed by heuristic A. In this case, the edge is labelled with the ratio $\frac{\sum_{G,II} R_B}{\sum_{G,II} R_A}$ where R_B is the number of registers (of considered type) computed by heuristic B and R_A is the number of registers computed by heuristic A.

The lattices are given on Figure 4.17, Figure 4.18, Figure 4.19 and Figure 4.20.

For instance, we read on Figure 4.17 that the number of registers of type BR computed by UAL heuristic is 1.069 greater than the number of registers of type BR computed by CHECK heuristic.

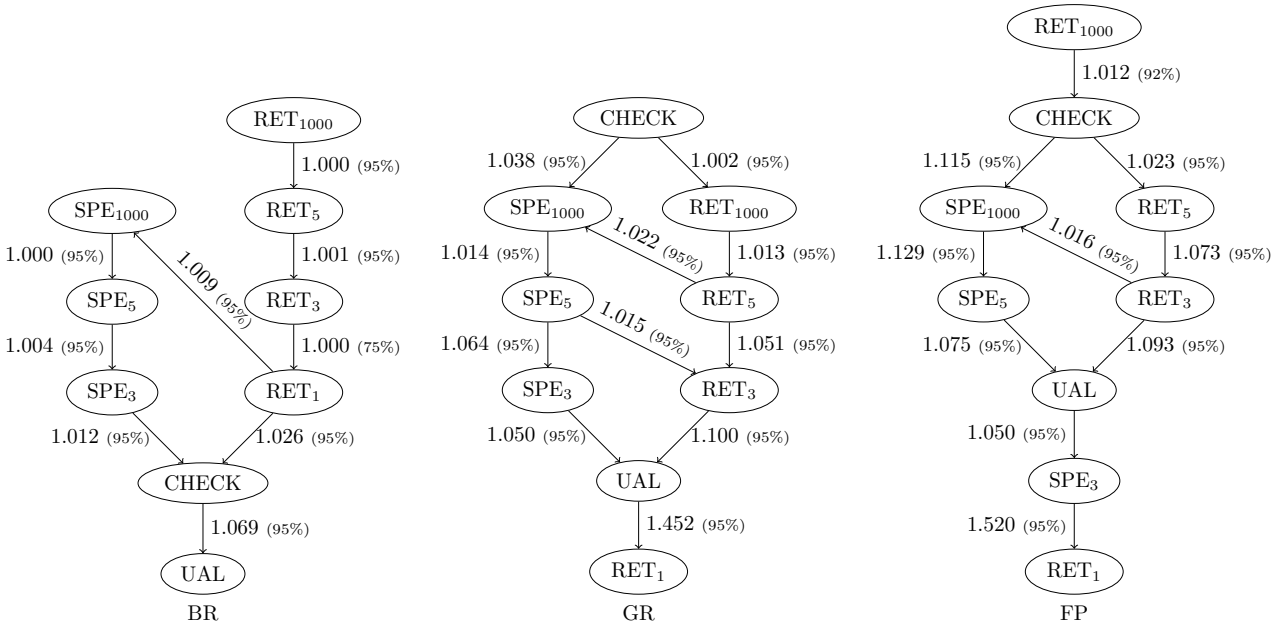


Figure 4.17: Comparison of the heuristics ability to reduce register pressure (SPEC2000)

Comments

Firstly, from these results, we observe that the ordering of the heuristics depends on the register type. Indeed, since the heuristics try to reduce register pressure of all types simultaneously, it happens that some performs better on one type that on the others. For instance, on the SPEC2000 set of benchmarks, RET_1 seems to produce good results on type BR but is the worst heuristic on the two other register types (GR and FP).

Secondly, we see that UAL and RET_1 are the two worst heuristics. Regarding UAL heuristic, this is not surprising since this is the most naive way to eliminate non-positive cycles. Regarding RET_1, this simply means that a single iteration of RET heuristic does not in general produce interesting results.

Thirdly, we see that RET heuristics seems to produce better results than SPE heuristics in terms of register requirement. But since we only have partial measures for RET heuristics, this conclusion should be taken with care. But nonetheless, it happens several times that RET_5 already produces better results than SPE_{1000}. So even if we increase the number of iterations for SPE, we are not sure to get something better than RET_5.

Finally, we observe that CHECK is sometimes the best heuristic (in particular for type GR and FP on all benchmarks except FFMPEG). We can explain this by the fact that the proportion of DDG with

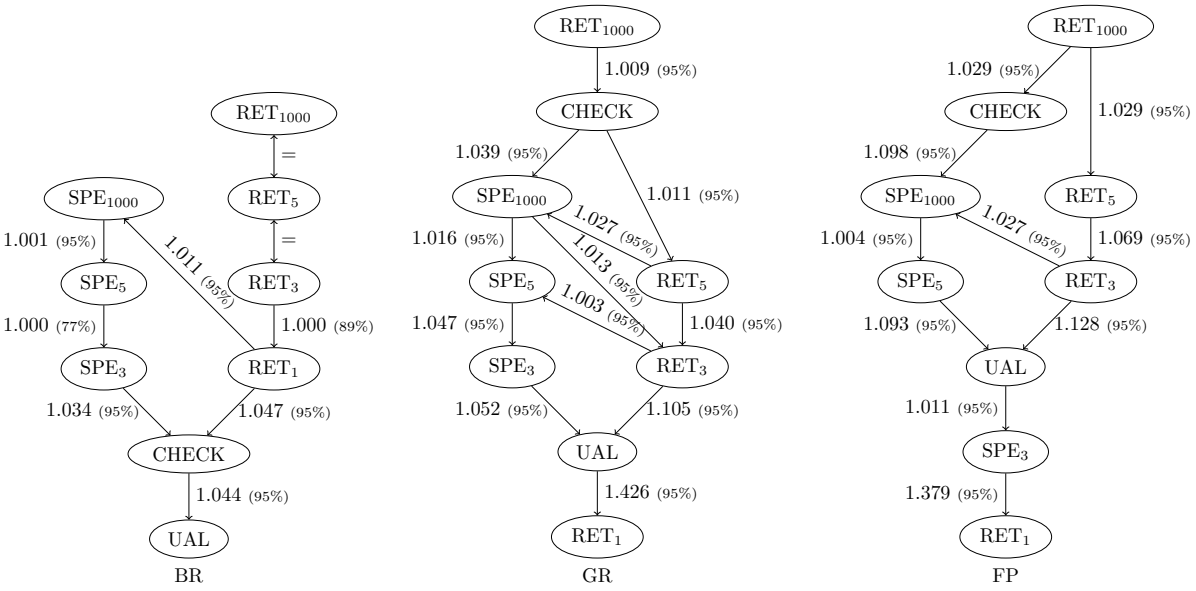


Figure 4.18: Comparison of the heuristics ability to reduce register pressure (MEDIABENCH)

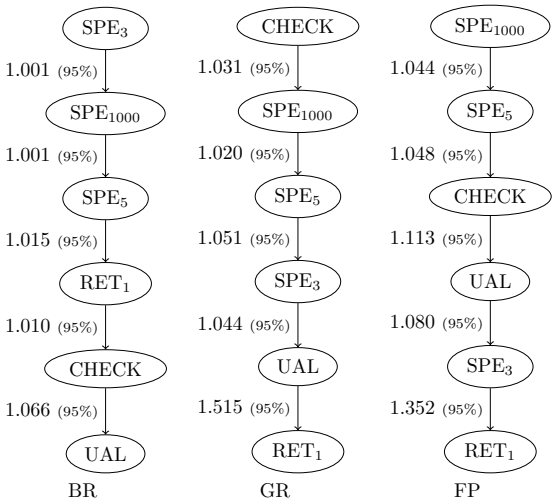


Figure 4.19: Comparison of the heuristics ability to reduce register pressure (SPEC2006)

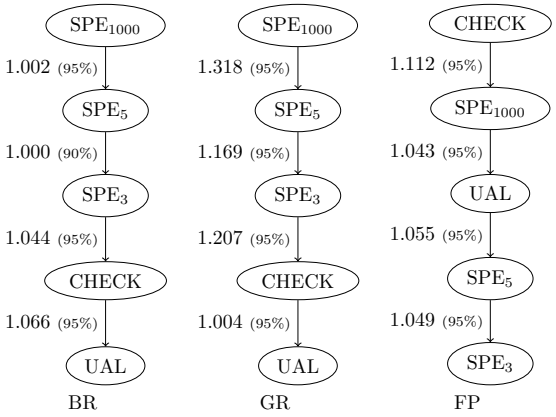
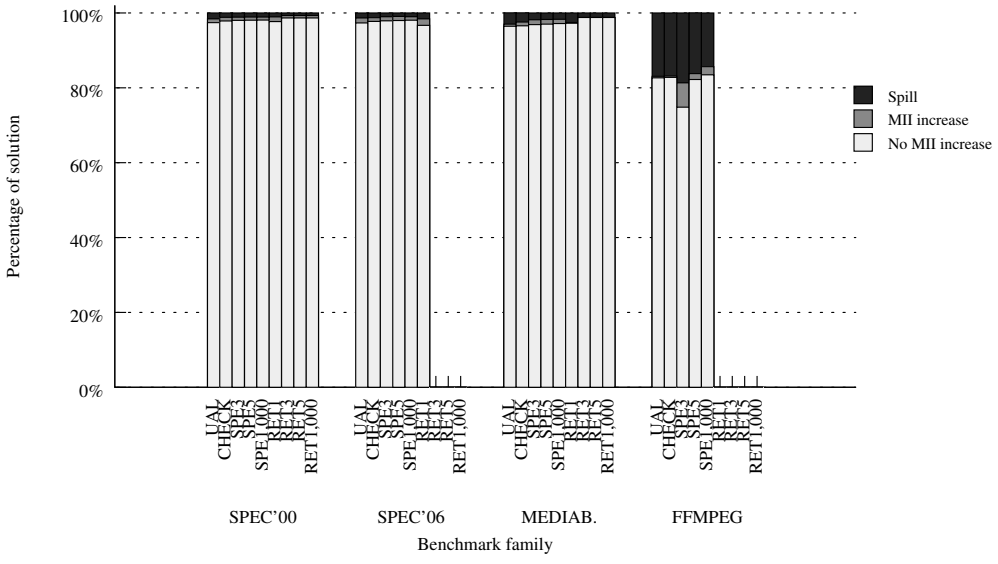


Figure 4.20: Comparison of the heuristics ability to reduce register pressure (FFMPEG)

non-positive cycles on SPEC2000, SPEC2006 and MEDIABENCH is low (less than 40%). Consequently, the reactive strategy (CHECK) is appropriate, since more than 60% of the DDG did not produce a non-positive circuit from the beginning.

4.3.2 Proportion of success when looking for a solution that satisfies the register constraints

Figure 4.21 shows the percentage of success produced by each heuristic. We decompose the solutions into three families: the DDG that have been solved without MII increase, the DDG that have been solved with MII increase, and the DDG that was not solved with SIRALINA (spilled).



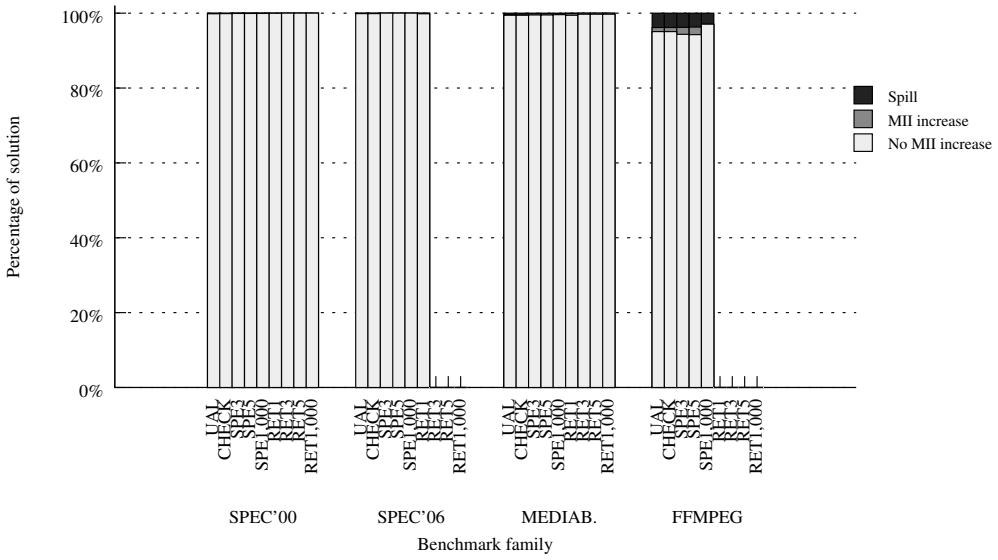
(a) architecture = small

Figure 4.21: Percentage of solution found by siralina and impact on the MII

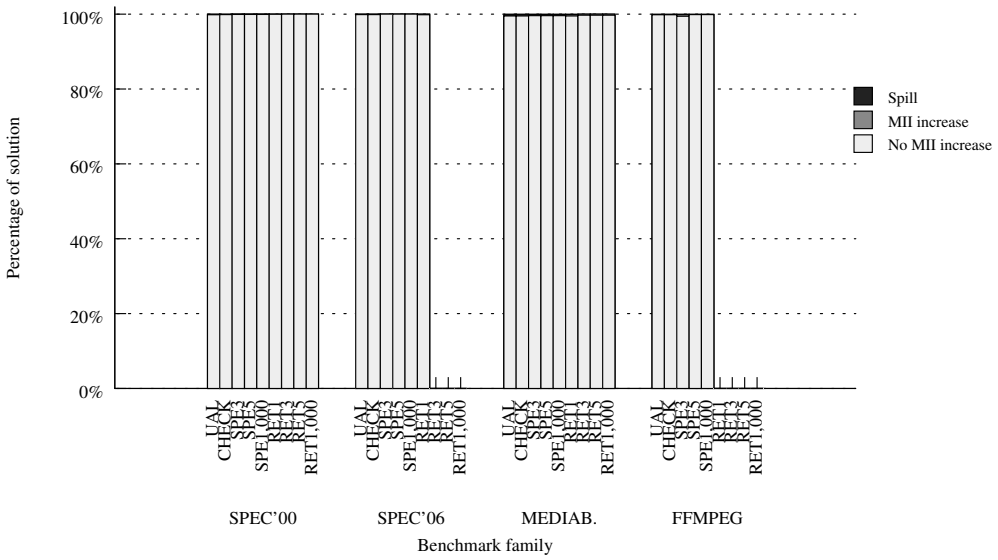
We note that SIRALINA found most of the time a solution that satisfies the register constraints. Of course, the percentage of success increased while the architecture constraints were relaxed.

Apart from the FFMPEG benchmarks under the small architecture constraints, the percentage of success is above 95%. In these cases, all the heuristics give comparable results.

For the FFMPEG benchmarks, we see that SPE_5 and SPE_{1000} give slightly better results than the naive heuristics (1 to 3% better).



(b) architecture = medium



(c) architecture = large

Figure 4.21: Percentage of solution found by siralina and impact on the MII (continued)

Observe finally that in most of the cases of success, the MII has not been increased at all.

4.3.3 Increase of the MII when looking for a solution that satisfies the register constraints

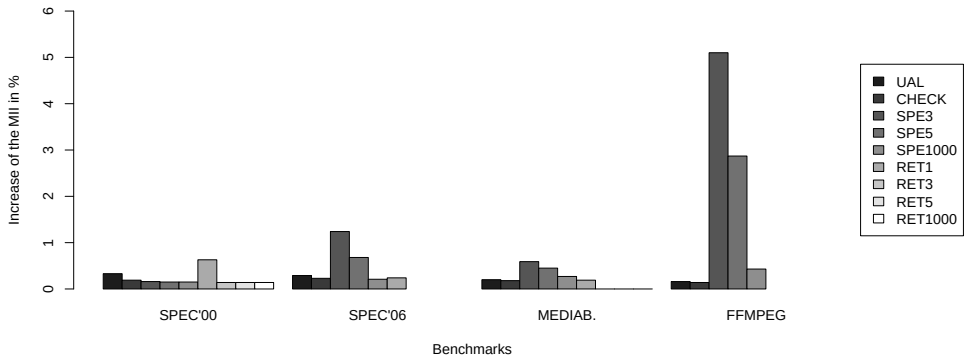
Figure 4.22 shows the increase of the MII compared to its initial value, after applying each of the variants of SIRALINA (). This increase is given by the formula $\frac{\sum \mathrm{MII}_h(G)}{\sum \mathrm{MII}(G)} - 1$, where $\mathrm{MII}_h(G)$ is the MII of the associated DDG computed by heuristic h. In other words MII_h is the smallest period II that satisfy the register constraints when w, use heuristic h. where $h \in \{\mathrm{UAL}, \mathrm{CHECK}, \mathrm{SPE}_n, \mathrm{RET}_n\}$. Remember that RET_n and SPE_n denote the Iterative SIRALINA when n is the maximal allowed number of iterations (however, the algorithm can stop before n iterations if a convergence).

These results show that the increase of the MII is very low (less than 6% in the worst case). It is clearly negligible on SPEC2000, SPEC2006 and MEDIABENCH benchmarks. On FFMPEG benchmarks, we see that when dealing with small architecture, SPE heuristics tends to increase the MII more than UAL or CHECK heuristics, whereas for bigger architecture, SPE_5 and SPE_{1000} gives slightly better results than UAL or CHECK.

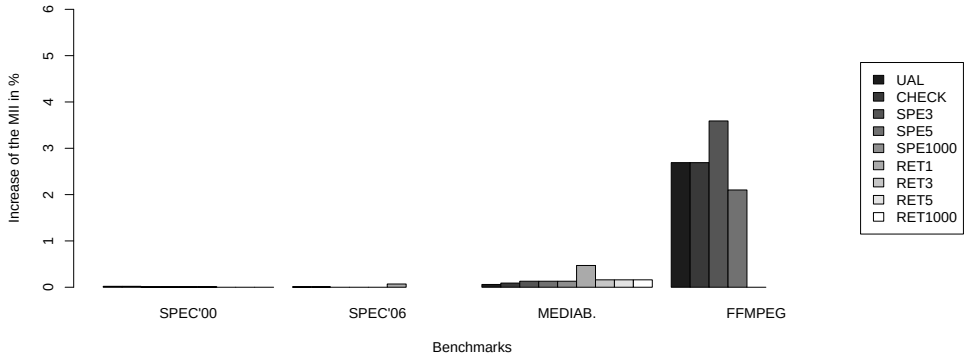
4.4 Conclusion

The conclusions we can take from this extensive experimental study are contrasted. On one hand, the results show that the two proactive heuristics RET and SPE allow to save a bit more of registers than the two naive heuristics UAL and CHECK. On the other hand, these results also show that our proactive heuristics are more expensive regarding the execution times than the reactive one. It has even appeared that the heuristics based on retiming ideas (i.e. RET heuristics) have a prohibitive cost, which makes them unusable in practice.

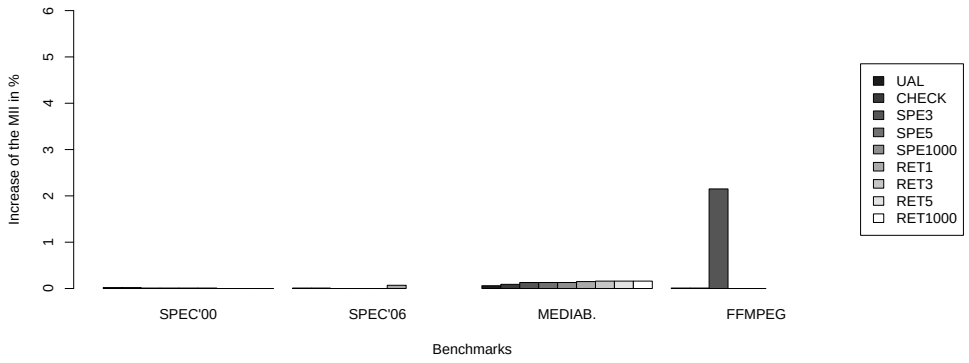
We thus advise the following policy. If the target architectures are embedded systems, where compilation time does not need to be interactive and where register constraints are strong, we advise to use SPE proactive heuristics. As we have seen, it optimises registers better than the reactive heuristic while being still quite cheap. On the contrary, if the target architecture is a general purpose computer (workstation, desktop, supercomputer), where register constraints are not too strong, it is probably sufficient to use the reactive heuristic CHECK as it already gives good results in practice and it is only between one and three times slower than UAL heuristic.



(a) architecture = small



(b) architecture = medium



(c) architecture = large

Figure 4.22: Increase of the MII

Chapter 5

Conclusion

Pre-conditioning a data dependence graph before SWP is a beneficial approach for reducing spill code and improving the performance of loops. Until now, schedule-sensitive register allocation was studied only for sequential and superscalar codes, with UAL code semantics.

When considering NUAL code semantics, the access to registers may be architecturally delayed. These delay accesses provide interesting compilation opportunities to save registers. These opportunities are exploited by the insertion of edges with non-positive latencies inside DDG.

Inserting edges with non-positive latencies inside DDG highlight two open questions to the community. First, existing software pipelining (SWP) and cyclic scheduling methods do not handle yet these non-positive latencies. Second, a pre-conditioning step that optimise registers before SWP may create cycles with non-positive distances.

DDG with non-positive distances have the drawback of not being lexicographic positive. This means that, when resource constraints are considered, the existence of a valid SWP is no longer guaranteed. This may cause the failure of the compilation process (no code is generated while the program is correct). Our experiments show that, if no care is taken, 30.77% of loops in SPEC2000 C applications induce non-lexicographic positive cycles (resp. 28.16%, 41.90% and 92.21 for SPEC 2006, MEDIABENCH and FFMPEG loops).

In order to avoid the situation of creating non lexicographic positive DDG, we studied in this report two strategies. First, we studied a reactive strategy that tolerates the problem: we start by optimising the register pressure at the DDG level without special care; if a non-positive cycle is detected, then backtrack and consider a UAL code semantics instead of NUAL; this means that we degrade the model of the architecture by not exploiting the opportunities offered by delayed accesses to registers. Second, we designed a proactive strategy that prevents the problem. The proactive strategy is an iterative process that increases the reuse distances until a fixed point is observed (or until we reach a limit in terms of iterations).

Concerning the efficiency of our strategies, the reactive strategy seems to perform well in practice in a regular compilation process: when the number of architectural registers is fixed, register minimisation is not necessary (just be sure to be below the architectural capacity). In this context, it is advised to not to try to prevent the problem, but to tolerate it in order to save compilation time. In other contexts of compilation, the number of architectural registers is not fixed. This is the case of reconfigurable circuits where the number of registers needed may be decided after code optimisation and generation. In such situation, our proactive strategy based on shortest path equations is very efficient in practice: the iterative register minimisation saves better registers than in the reactive strategy, while the compilation time stays reasonable (though greater than the reactive strategy). We found that 5 iterations are sufficient in general to have satisfactory results.

Finally, for the reproducibility of our results, a new version of `SIRAlib` that includes the reactive and proactive strategies is released under LGPL license.

Bibliography

[1] Alix Munier. A graph-based analysis of the cyclic scheduling problem with time constraints: schedulability and periodicity of the earliest schedule. *Journal of Scheduling*, 2010. To appear (accepted for publication).

[2] Benoît Dupont-de-Dinechin. Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates. In *LCPC '96: Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, pages 231–245, London, UK, 1997. Springer-Verlag.

[3] Sébastien Briais and Sid-Ahmed-Ali Touati. Schedule-sensitive register pressure reduction in innermost loops, basic blocks and super-blocks. Technical Report HAL- 00436348, University of Versailles Saint-Quentin en Yvelines, November 2009. http://hal.archives-ouvertes.fr/inria-00436348.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[5] Karine Deschinkel and Sid-Ahmed-Ali Touati. Efficient method for periodic task scheduling with storage requirement minimization. In *Proceedings of 2nd Annual International Conference on Combinatorial Optimization and Applications (COCOA 2008)*, LNCS, Saint Johns, Newfoundland, Canada, August 2008. Springer.

[6] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[7] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[8] Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.

[9] Sid-Ahmed-Ali Touati. *Register Pressure in Instruction Level Parallelism*. PhD thesis, Université de Versailles Saint-Quentin, 2002.

[10] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 2004.

Appendix A

Results for RET heuristic with a time-out enabled

We have seen in Chapter 4 that RET heuristic is very slow in practice, and for this reason we get partial results about it. We report on the table below the total amount of computation time that we spent for each heuristic. We also indicate the percentage of completion of the computation: it is simply the ratio between the number of computations performed per DDG (forall possible II) and the total number of computations that were to be done. We say that a computation cannot be done when we stop the integer linear solver because of a long processing time (multiple weeks of waiting for instance). A computation is said completed when RET returns a solution for a fixed value of II.

Heuristic	Total time	Percentage of completion
UAL	$\approx$ 7 hours	100%
CHECK	$\approx$ 18 hours	100%
SPE_3	$\approx$ 39 hours	100%
SPE_5	$\approx$ 58.5 hours	100%
SPE_{1000}	$\approx$ 189 hours	100%
RET_1	$\approx$ 528.4 hours	25.1%
RET_3	$\approx$ 47.6 hours	8.6%
RET_5	$\approx$ 187.6 hours	8.6%
RET_{1000}	$\approx$ 203.3 hours	8.6%

In order to study more deeply the behavior of RET heuristic, we thus have modified it and added a timeout of 10 seconds over each mixed integer linear program resolution. 10 seconds is an arbitrary time-out, it is a reasonable compilation time that can be devoted to embedded systems for instance. We could use another value for the time-out of course. Note that sine solving the mixed integer linear program of RET is an NP-complete problem, reaching the optimality is not tractable in practice.

The time we spent for this new set of experiments is reported below.

Heuristic	Total time	Percentage of completion
RET_3	$\approx$ 383.3 hours	40.7%
RET_5	$\approx$ 433.6 hours	40.7%
RET_{1000}	$\approx$ 481 hours	40.7%

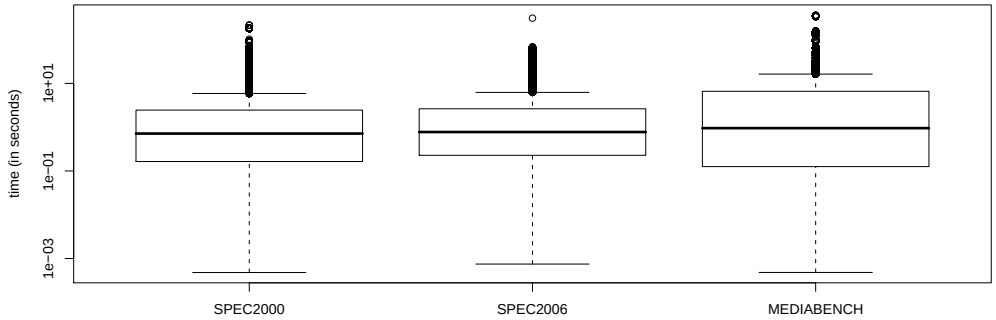
We report below the experimental results we obtained with this modified heuristic. So the experimental results we report in this section are only done on a subset of the DDG.

A.1 Execution times

A.1.1 Time to reduce register pressure for a fixed II

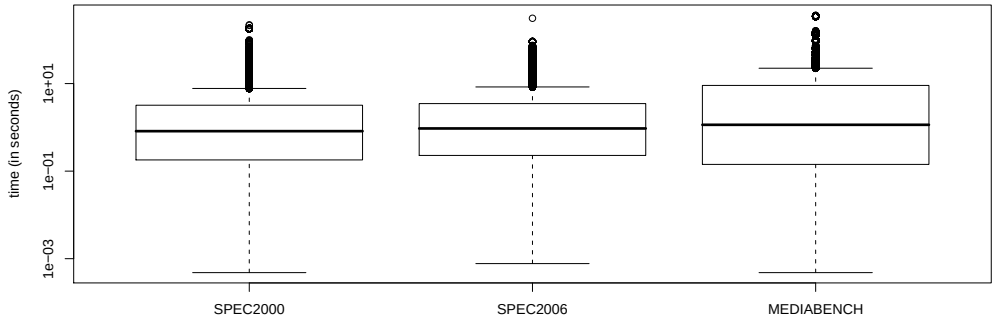
In this section, we solve RETn SIRALINA with all values of II. Figure A.1 shows the distribution of execution times of RET_n heuristic for $n \in \{3, 5, 1000\}$ and for all values of II.

Again, we were unable to get a full set of results. In particular, we abandoned the study of FFMPEG benchmarks. For the other family of benchmarks, we got however results for almost all DDGs.



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000478	0.000746	0.00048
FST	0.164101	0.227695	0.125581
MEDIAN	0.713944	0.774588	0.949614
THD	2.45425	2.62851	6.60028
MAX	215.101	309.27	361.062

(a) 3 iterations

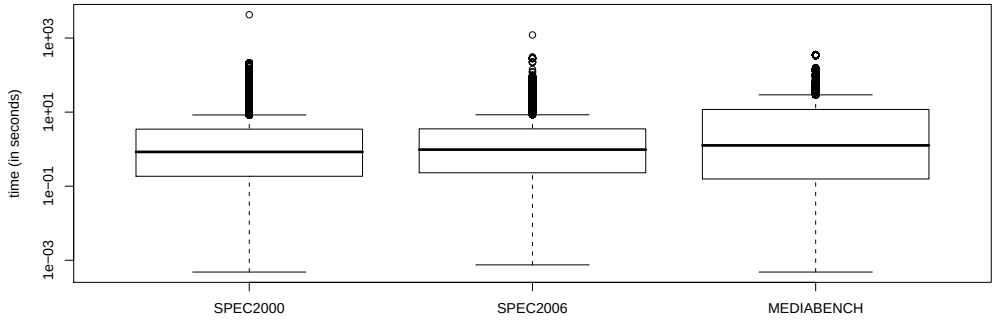


	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.00048	0.000772	0.00048
FST	0.180191	0.228423	0.142491
MEDIAN	0.818336	0.943738	1.14341
THD	3.20334	3.48971	9.07013
MAX	220.249	310.692	362.73

(b) 5 iterations

Figure A.1: Execution times of RET (in seconds)

The results about RET heuristic do not modify our initial diagnosis. On the (almost) full set of benchmarks, we observe that the median time is slightly greater than what we observed in Chapter 4 for SPEC2000 and much bigger for MEDIABENCH benchmarks (for which we only had a really small fraction of results previously).



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000481	0.000752	0.000481
FST	0.185359	0.230511	0.156693
MEDIAN	0.841932	0.972459	1.26352
THD	3.47008	3.53216	11.7971
MAX	4257.69	1212.11	359.252

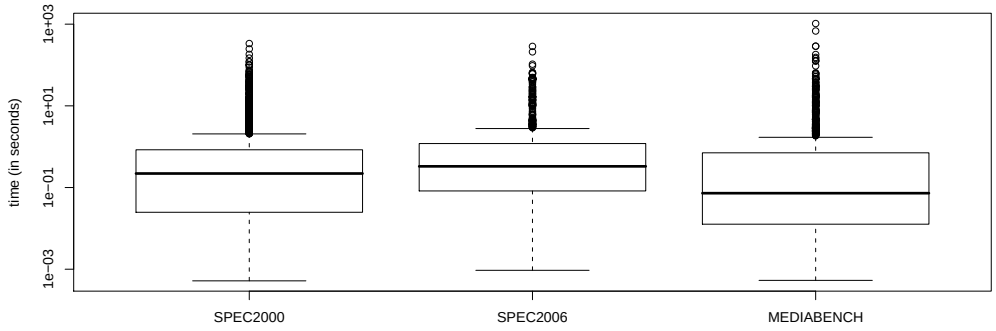
(c) 1000 iterations

Figure A.1: Execution times of RET (in seconds)

A.1.2 Time to reduce register pressure below the architectural capacity

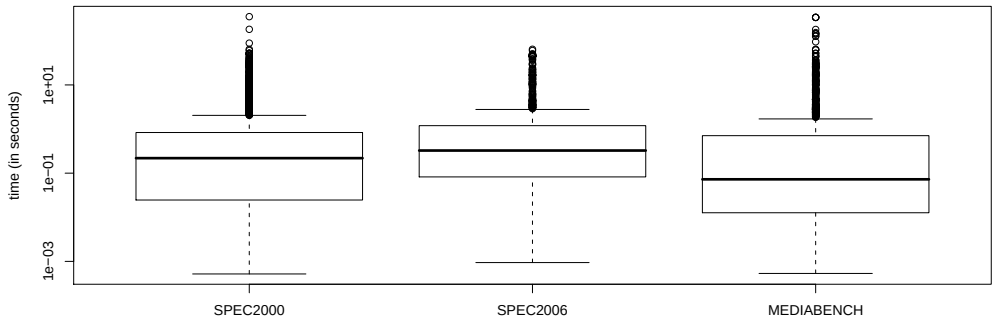
When the number of available registers is fixed in the architecture, RETn may iterate over multiple values of II to be compute a solution below the number of available registers.

Figure A.2 (resp. Figure A.3, Figure A.4) shows the distribution of execution times of RET_3 (resp. RET_5, RET_{1000}).



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000519	0.00094	0.000533
FST	0.024669	0.082421	0.012661
MEDIAN	0.220098	0.326338	0.072678
THD	0.840102	1.1932	0.707905
MAX	335.705	285.9231	1036.443

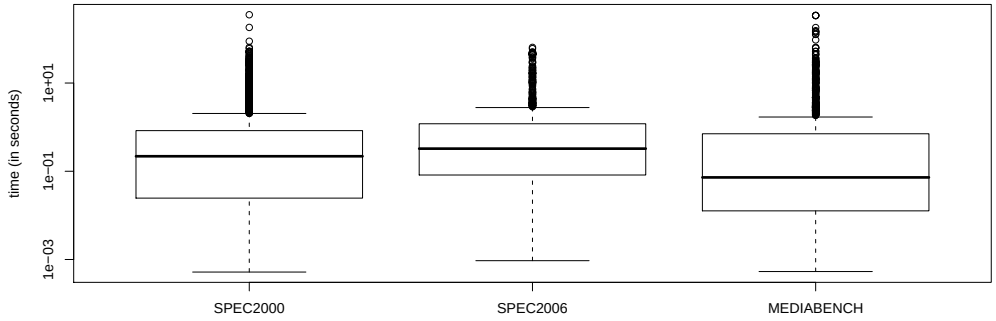
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000519	0.00094	0.000533
FST	0.024559	0.082068	0.012661
MEDIAN	0.218747	0.325341	0.072506
THD	0.833685	1.19307	0.707905
MAX	353.3983	64.3315	339.8613

(b) medium architecture

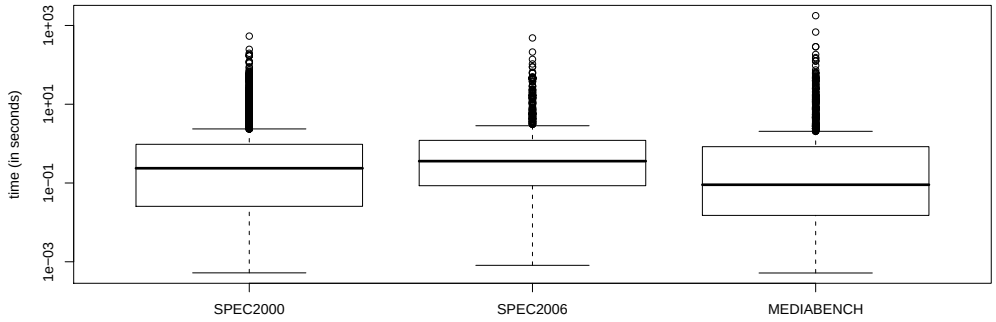
Figure A.2: Execution times of RET_3 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000519	0.00094	0.000533
FST	0.024559	0.082068	0.012661
MEDIAN	0.218747	0.325341	0.072506
THD	0.833685	1.19307	0.707905
MAX	353.3983	64.3315	339.8613

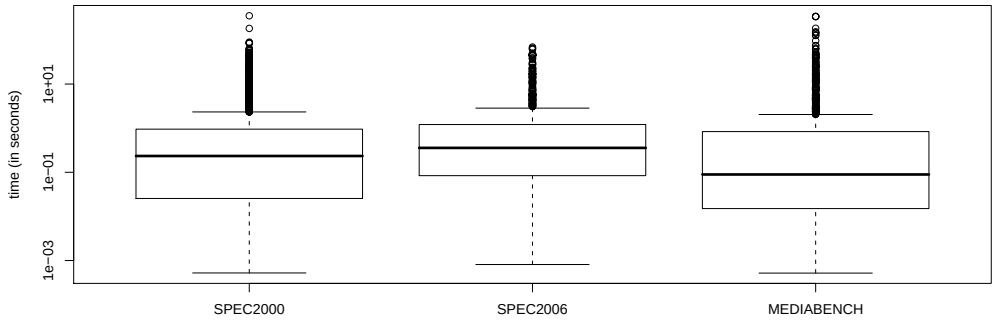
(c) large architecture

Figure A.2: Execution times of RET_3 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000523	0.000812	0.000519
FST	0.025526	0.085177	0.015047
MEDIAN	0.236213	0.358309	0.090279
THD	0.960228	1.20872	0.836207
MAX	536.4077	482.4372	1772.849

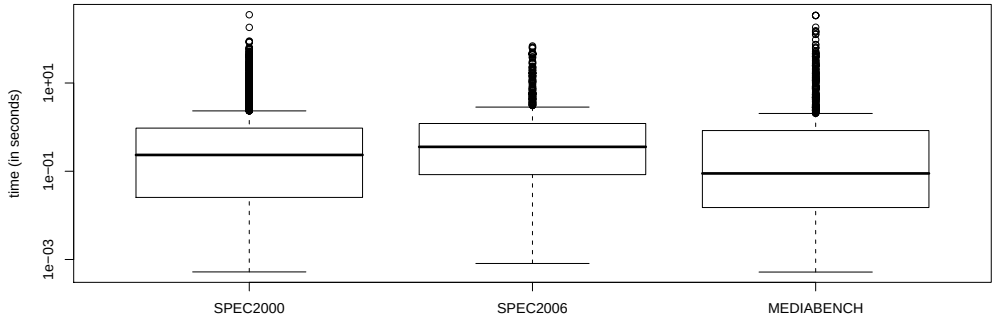
(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000523	0.000812	0.000519
FST	0.025499	0.082581	0.015047
MEDIAN	0.234029	0.35571	0.089192
THD	0.949672	1.20777	0.836207
MAX	352.7015	69.0297	340.1572

(b) medium architecture

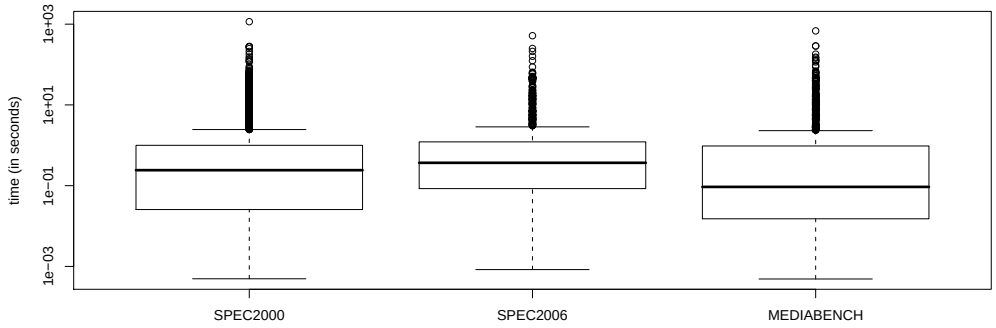
Figure A.3: Execution times of RET_5 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000523	0.000812	0.000519
FST	0.025499	0.082581	0.015047
MEDIAN	0.234029	0.35571	0.089192
THD	0.949672	1.20777	0.836207
MAX	352.7015	69.0297	340.1572

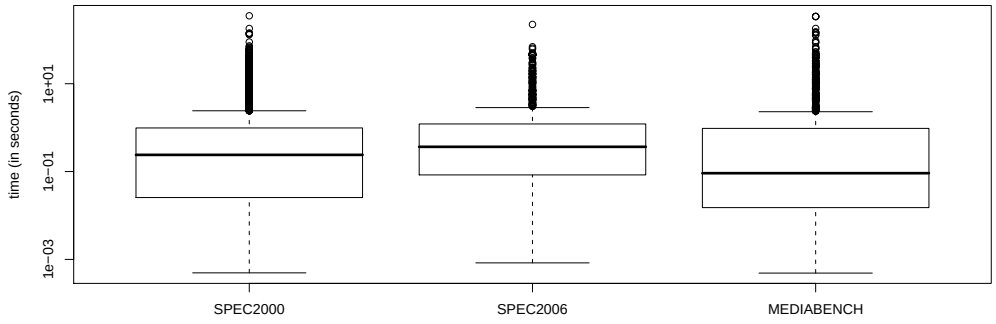
(c) large architecture

Figure A.3: Execution times of RET_5 (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000494	0.000834	0.000488
FST	0.025619	0.084362	0.015112
MEDIAN	0.241527	0.368236	0.093246
THD	0.997599	1.21443	0.961941
MAX	1152.644	517.8463	681.305

(a) small architecture



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000494	0.000834	0.000488
FST	0.025611	0.083426	0.015112
MEDIAN	0.239996	0.363658	0.091823
THD	0.9836	1.21189	0.961217
MAX	352.3916	223.799	340.286

(b) medium architecture

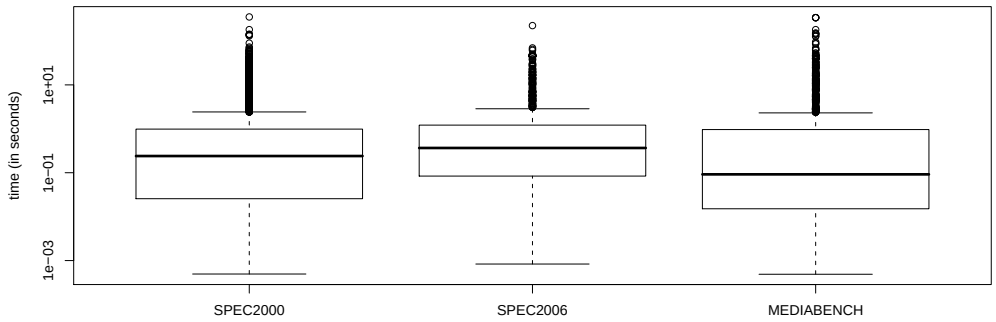
Figure A.4: Execution times of RET_{1000} (in seconds)

Again, the above results just confirm the analysis we made previously in Chapter 4: RET heuristics are not usable in practice.

A.1.3 Convergence in terms of number of iterations of Algorithm 1

Figure A.5 shows the distribution of the number of iterations of Algorithm 1 before convergence when using RET heuristic (maximal allowed number of iterations is truncated at 1000). RETn is executed for all values of II.

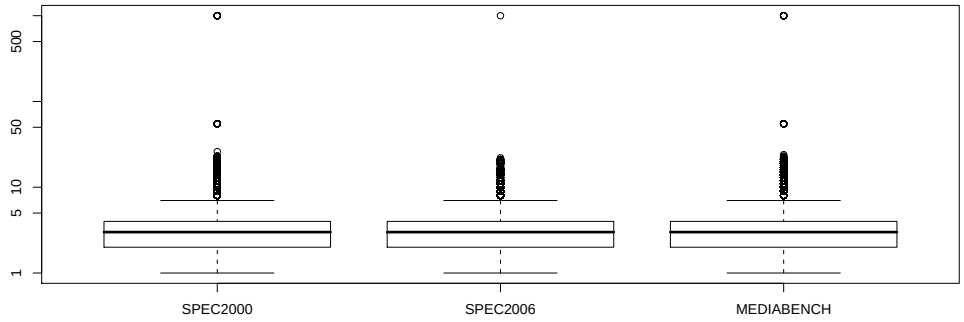
These results just show that in term of number of iterations, RET heuristics converges faster than SPE heuristics. In 75% of the cases, 4 iterations are sufficient.



	SPEC2000	SPEC2006	MEDIABENCH
MIN	0.000494	0.000834	0.000488
FST	0.025611	0.083426	0.015112
MEDIAN	0.239996	0.363658	0.091823
THD	0.9836	1.21189	0.961217
MAX	352.3916	223.799	340.286

(c) large architecture

Figure A.4: Execution times of RET_{1000} (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH
MIN	1	1	1
FST	2	2	2
MEDIAN	3	3	3
THD	4	4	4
MAX	1000	1000	1000

Figure A.5: Maximum number of iterations for RET

However, we also observe that in a few cases, the upper bound of 1000 iterations is reached. This possibly means that, as SPE heuristics, RET heuristics may not terminate in rare few cases (if the number of iterations was not bounded).

A.2 Qualitative analysis

A.2.1 Number of saved registers

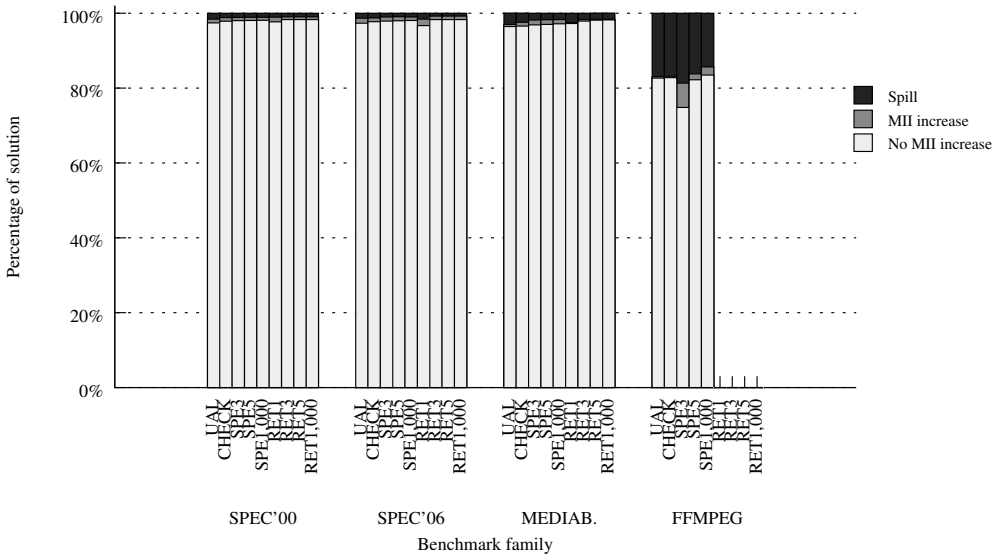
We indicate in the table below, for each benchmark family, and each register type, the ratios $\frac{\sum_{G,II} R_{UAL}}{\sum_{G,II} R_n}$ where R_n is the number of registers (of considered type) computed by heuristic RET_n (with $n \in \{3, 5, 1000\}$) and R_{UAL} is the number of registers computed by heuristic UAL. The experiments are done on all values of II.

Type	Heuristic	SPEC2000	SPEC2006	MEDIABENCH
BR	RET_3	1.095	1.106	1.079
	RET_5	1.095	1.106	1.080
	RET_{1000}	1.095	1.106	1.080
GR	RET_3	1.108	1.116	1.153
	RET_5	1.161	1.156	1.191
	RET_{1000}	1.177	1.167	1.211
FP	RET_3	1.286	1.144	1.278
	RET_5	1.369	1.288	1.321
	RET_{1000}	1.450	1.346	1.350

These results show that RET heuristics is really efficient for reducing the register pressure. It clearly outperforms UAL, CHECK and SPE heuristics.

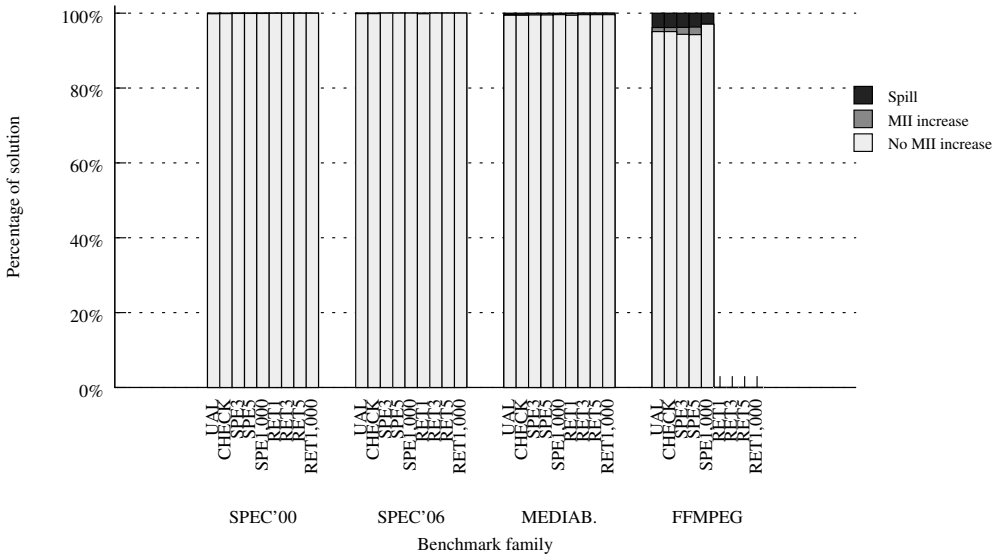
A.2.2 Percentage of success when looking for a solution that satisfies the register constraints

Figure A.6 shows the percentage of solutions found by each heuristic and the percentage of DDG that need spilling. It also shows, in the case where a solution to the SIRA problem exists, whether the MII has been increased or not.

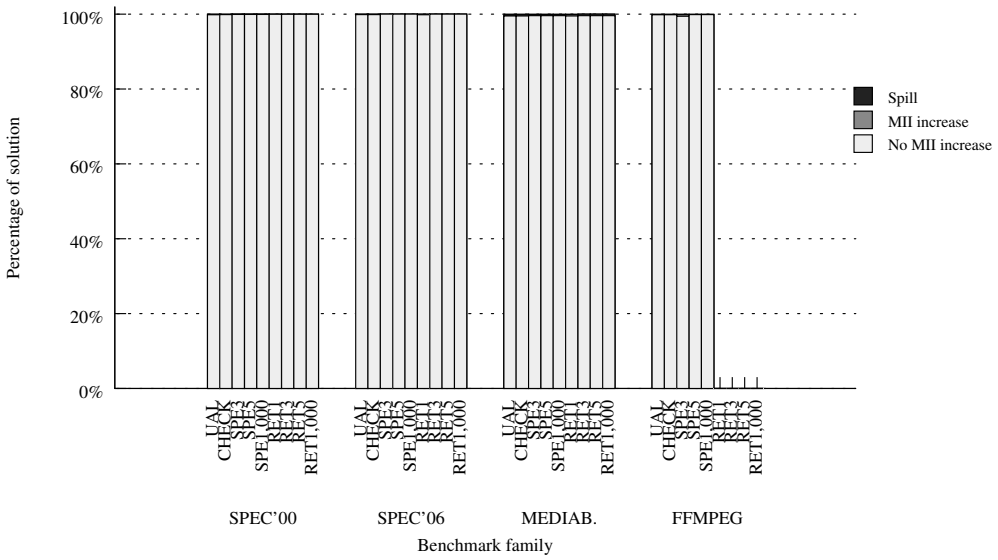


(a) architecture = small

Figure A.6: Percentage of solution found by siralina and impact on the MII



(b) architecture = medium



(c) architecture = large

Figure A.6: Percentage of solution found by siralina and impact on the MII (continued)

We see that RET heuristics find most of the time a solution that satisfies the register constraints. It is often better than the other heuristics. Also, the MII is rarely increased.

A.2.3 Increase of the MII

This increase is given by the formula $\frac{\sum \mathrm{MII}_h(G)}{\sum \mathrm{MII}(G)} - 1$, where $\mathrm{MII}_h(G)$ is the MII of the associated DDG computed by heuristic h. In other words MII_h is the smallest period II that satisfy the register constraints when w, use heuristic h. where $h \in \{\mathrm{RET}_3, \mathrm{RET}_5, \mathrm{RET}_{1000}\}$. The table below shows the increase of the MII when RETn found a solution.

Architecture	Heuristic	SPEC2000	SPEC2006	MEDIABENCH
Small	RET_3	0.14	0.21	0.16
	RET_5	0.14	0.21	0.15
	RET_{1000}	0.14	0.21	0
Medium	RET_3	0.01	0	0.14
	RET_5	0.01	0	0.14
	RET_{1000}	0.01	0	0.14
Large	RET_3	0.01	0	0.14
	RET_5	0.01	0	0.14
	RET_{1000}	0.01	0	0.14

As we previously observed, the increase of the MII is very low, and often small than the one obtained with the other heuristics

Conclusion

The modified version of RET heuristics that incorporate a time-out (10 seconds) allowed us to obtain additionnal results. These results have shown that RET family of heuristics are very good for reducing register pressure of DDGs. Unfortunately, the execution times remained prohibitive. It is thus unfortunately not realistic to consider using RET heuristics in practice.

UFR des sciences	:	45 avenue des Etats Unis. 78035 Versailles cedex
IUT de Velizy et de Rambouillet	:	10-12 avenue de l'Europe. 78140 Vélizy.
UFR des Sciences Sociales et des Humanité	:	47 boulevard Vauban. 78047 Guyancourt cedex
Faculté de droit et de science politique	:	3, rue de la Division Leclerc. 78280 Guyancourt
IUT de Mantes en Yvelines	:	7 rue Jean Hoët - 78200 Mantes la Jolie
UFR de Médecine Paris-Ile-de-France Ouest	:	9 boulevard d'Alembert Bâtiment François Rabelais. 78280 Guyancourt
Institut des Langues et des Etudes Internationales	:	5-7, boulevard d'Alembert. 78280 Guyancourt
Institut des Sciences et Techniques des Yvelines	:	45 avenue des Etas Unis - 78035 Versailles cedex
Observatoire des Sciences de l'Univers de l'UVSQ	:	11 boulevard d'Alembert. 78280 Guyancourt

Éditeur
UVSQ - Siège : 55 avenue de Paris - 78035 Versailles Cedex (France)
http://www.uvsq.fr